IMPERIAL COLLEGE OF SCIENCE, TECHNOLOGY AND MEDICINE
University of London

A Statistical Examination of The Evolution of the UNIX System

by

Shamim Sharifuddin Pirzada

September 1988

A thesis submitted for the Degree of **Doctor of Philosophy** of the University of London and for the Diploma of Membership of the Imperial College.

ABSTRACT

The UNIX system is one of the most successful operating systems in use today. However, due to its age, and in view of the tendencies of other operating systems to degenerate over time, concern has been expressed about its potential for further evolution. Modelling techniques have been proposed to view and predict the evolution of software but they have not yet been sufficiently evaluated.

The project uses one such technique, developed by Lehman and others, to examine the evolution of UNIX and attempt a prognosis for its future. Hence it critically evaluates Lehman's concepts of program evolution.

A brief survey of quantitative software modelling techniques is given with particular emphasis on models which predict the behaviour of software systems *already in use*. The development of Lehman's "Theory of Program Evolution" is reviewed and the implications of the hypotheses proposed in the theory are discussed.

Also, the history of UNIX is presented as a sequence of releases from the main UNIX centres in the Bell System and the University of California, Berkeley.

An attempt is made to construct statistical models of the UNIX evolution process by plotting the progress of the three main branches of the UNIX evolution tree (Research UNIX, the System V stream and BSD/UNIX) in terms of changes in various system and process attributes such as size, growth-rate, work-rate and staffing.

The examination reveals that none of the branches of UNIX are suffering structural degradation to the same extent as, for instance, IBM's OS/360. However, the supported and commercial stream does show an upwards trend in system complexity *since commercialization*. Furthermore, the plots show a marked difference in the behaviour of the three systems and permit numerical predictions, though not statistically significant, to be made for only the System V stream.

The effect of the environment (in research, commercial and academic programming cultures) on the dynamics of the programming process is investigated. This suggests that processes in a strongly commercial environment are much more likely to exhibit structural deterioration and statistically smooth evolution patterns than processes in pure research environments.

ACKNOWLEDGEMENTS

I would like to thank Prof. Manny Lehman for guiding me through the initial stages of this project and Dr. John Jenkins for expertly managing the closing stages. Their supervision has been invaluable to me.

The constant encouragement of Dr. Doug McIlroy, my supervisor at Bell Labs, kept this project going. I would like to thank him, and the rest of Center 1127, for their help.

I would also like to thank Prof. D. Ferrari for arranging my UC Berkeley visit and Kirk McKusick for his help during my stay there.

A number of people lent me the various manuals and distribution tapes which made this study possible, I would like to thank them all.

My thanks also goes out to a number of people at Imperial Software Technology who supported and encouraged me in the final stages.

Finally I would like to thank my wife Seema, my parents and the rest of my family who endured 'the thesis' for far too long.

This research was partially sponsored by the Science and Engineering Research Council of the U.K.

TRADEMARKS

UNIX is a registered trademark of AT&T in the USA and other countries.
DEC is a registered trademark of Digital Equipment Corporation.
VAX is a registered trademark of Digital Equipment Corporation.
IBM is a registered trademark of International Business Machines
POSIX is a registered trademark of the Institute of Electrical and Electronic Engineers.
ICL is a registered trademark of International Computers Limited.
NCR is a registered trademark of National Cash Registers.
Xenix is a registered trademark of Microsoft.
SunOS is a registered trademark of Sun Microsystems, Inc.

CONTENTS

1 INTRODUCTION 8
1.1 WHY UNIX? 8
1.2 THE PROBLEM 10
1.3 OBJECTIVES OF THE PROJECT 12
1.4 OVERVIEW OF THE THESIS 13

2 REVIEW OF PROGRAM EVOLUTION CONCEPTS 14
2.1 INTRODUCTION 14
2.2 COST ESTIMATION TECHNIQUES 14
2.3 A REVIEW OF PROGRAM EVOLUTION DYNAMICS 16
2.3.1 Origins 16
2.3.2 The 'laws' of Program Evolution 16
2.4 EXPECTED BEHAVIOUR 18
2.4.1 The measurements 18
2.4.2 The evolution pattern 18
2.5 THE SPE CLASSIFICATION AND OTHER IDEAS 24
2.6 EMPIRICAL EVIDENCE 26
2.7 CRITICAL EVALUATIONS 26
2.8 NEEDS 27

3 THE HISTORY OF UNIX 28
3.1 INTRODUCTION 28
3.2 ORIGINS 29
3.3 THE UNIX RELEASE TREE 31
3.3.1 Research 33
3.3.2 Unix Support Group, UNIX Development Laboratory & AT&T-IS 35
3.3.3 Programmer's Workbench 41
3.3.4 Columbus Operations Systems Group 43
3.3.5 University of California at Berkeley 44
3.4 LICENSING 46

4 PROJECT METHODOLOGIES 49
4.1 INTRODUCTION 49
4.2 SURVEY OF METRIC MEASURES 49
4.2.1 Size 50
4.2.2 Complexity 53
4.2.3 Process Attributes 55
4.2.4 Other Attributes 56
4.3 SOURCE MATERIAL 58
4.3.1 Ideal sources 58
4.3.2 Configuration Management in UNIX Groups 59
4.3.3 Data Collection: The Project Database 63
4.4 DATA EXTRACTION 65
4.4.1 Devices 66
4.4.2 Loading 66

4.4.3 Reading 67
4.5 SOURCE CODE STRUCTURE 67
4.5.1 UNIX directory hierarchy 68
4.5.2 The 'C' language peculiarities 69
4.5.3 Program structure 71
4.5.4 Program Construction 72
4.6 MEASUREMENT OF THE METRICS 72
4.6.1 Preprocessing 72
4.6.2 Size 74
4.6.3 Complexity 75
4.6.4 Work-rate 76
4.6.5 Increased Complexity 77
4.6.6 Release Content 77
4.6.7 Others 77
4.7 CHOICE FOR UNIX MODELS 77

5 MODELS OF THE UNIX EVOLUTION PROCESS 79
5.1 INTRODUCTION 79
5.2 THE PLOTS 80
5.2.1 Size 81
5.2.2 Module Inter-connectivity 86
5.2.3 Release Interval 90
5.2.4 Work-rate 93
5.2.5 Growth 98
5.2.6 Documentation 103
5.3 PROGNOSIS FOR THE FUTURE OF UNIX 106
5.3.1 Necessary conditions for the predictions 106
5.3.2 The Research Stream 107
5.3.3 The Academic Stream 107
5.3.4 The Supported and Commercial Stream 107
5.3.5 Recommendations 108
5.4 EVALUATION OF THE THEORY OF PROGRAM EVOLUTION 109
5.4.1 Characteristics expected by the 'laws' 109
5.4.2 Behaviour displayed by the UNIX systems 110
5.4.3 Statistically Smooth Behaviour 111
5.4.4 Conclusions 111

6 PROGRAMMING CULTURES 113
6.1 INTRODUCTION 113
6.2 ENVIRONMENTAL FACTORS EFFECTING THE PROGRAMMING PROCESS 113
6.2.1 Measures of Quality 114
6.2.2 Feedback mechanisms 115
6.2.3 Release Mechanisms 116
6.2.4 Change mechanisms and strategies 118
6.2.5 The Product 120
6.2.6 Staffing, organization and management 122
6.3 CONCLUSIONS 124

7 UNDER PRESSURE 127
7.1 INTRODUCTION 127
7.2 CULTURAL ASPECTS OF THE PROCESS 128
7.2.1 Test against known behaviour 128
7.2.2 Recommendations for further work 129
7.3 PROPOSALS FOR THEORY OF PROGRAM EVOLUTION 129
7.3.1 Continuing Growth 130
7.3.2 Increasing complexity? 130
7.3.3 Smooth behaviour in commercial systems 130
7.3.4 Recommendations for further work 131
7.4 ARE METRIC MODELLING TECHNIQUES USEFUL? 131
7.5 THE UNIX MARKET PLACE: Survival of the fittest? 132
7.5.1 The past 132
7.5.2 The present 133
7.6 CLOSING REMARKS 134

A GLOSSARY 136

B REFERENCES 138

C MISCELLANEOUS PLOTS 147
C.1 INTRODUCTION 147
C.2 SIZE 147
C.3 PLOTS REQUIRING THE PARSER 147

D PROGRAM LISTINGS 151
D.1 INTRODUCTION 151
D.2 MECHANICS OF THE ANALYSIS 151
D.2.1 Generating File List 151
D.2.2 Analysing each file 151
D.2.3 Calculating metrics for the whole release 152
D.2.4 Plotting the results 153
D.3 LISTINGS 153
D.3.1 Parser 153
D.3.2 AWK scripts 179

Chapter 1
INTRODUCTION

"Major technological breakthroughs, like the transistor, are rare events. These breakthroughs have far-reaching effects on science, business, and, at times, society. The UNIX operating system is such a breakthrough."

(R. L. MARTIN, *The UNIX System*, 1984)

1.1 WHY UNIX?

UNIX is one of the most successful operating systems in use today. Free from commercial pressures in its early stages, the system has been able to pursue technical excellence without interference from marketing departments and profit conscious managers. Analysts have estimated that the UNIX market will be worth more than US $11 billion by 1991.[1]

UNIX expertise

As a result of its technical excellence, almost every major educational institute in the world uses the UNIX system, both as a computing resource and as an example of good operating system design and implementation. This has resulted in a generation of programmers being 'brought up' on UNIX. This programmer body, estimated at over 250,000 in the USA alone [DAS88] (more than any other system), demand UNIX where ever they go. These programmers have been able to obtain expertise in UNIX because the system's simplicity and elegance of design and implementation has been visible from the start, in the form of supplied on-line source code.[2]

---

1. Reported in *Computing* May 1988.
2. This practise has only recently been changed.

Indeed, its influence can be seen in other operating systems developed since. Features such as hierarchical file systems, programmable high level command interpreters and pipes are much more common place now than they were before UNIX.

Hardware spectrum

UNIX was one of the first operating systems to be written[3] in a high level language, traditionally they were written in assembler. As a direct result of which, it was the first to be ported to a different machine. There were no political obstructions to this as the researchers who wrote UNIX (and carried out the port) were not working for a hardware manufacturer. Since then, it has been ported to a number of other machines and now the demand for UNIX is such that *every* major hardware manufacturer (including IBM, DEC, HP, ICL, UNISYS, CDC, Data General and Apple) has, or is shortly going to have, a version of UNIX for their machines. In addition, a number of companies have designed their hardware to be purely UNIX engines (e.g. AT&T, Sun and Pyramid). Even the established manufacturers are bringing out machines which are designed from the start to run UNIX (IBM: PC-RT; CRAY: CRAY-3). Hence, UNIX is now available on over 200 lines of computers, ranging from micros to the largest supercomputers. And this list is ever increasing.

Third party support

Initially, UNIX was offered without warranty or support. But with its well written, high level source code provided on-line, other organisations were able to tailor the system to meet their own needs. As a result several companies gained expertise in the system's internals and when AT&T offered sub-licenses, allowing third parties to modify the system and sell their own guarantee and support, several companies jumped at it. Currently, various flavours of UNIX can be had from a number of software houses (the most famous of which is MicroSoft's Xenix). Indeed, a number of different UNIX systems can be had *for the same machine*!

Applications vehicle

The high degree of portability offered by UNIX allows applications built on top of it to be portable as well. This has a strong commercial advantage in that an application does not have to be discarded at every hardware upgrade and is not tied in to a particular vendor. As a result, a

3. More accurately: re-written.

number of software houses are developing their products on top of UNIX while others are converting theirs to UNIX. In 1984, it was estimated that over 100 companies in the US were doing so [MAR84], that figure has probably gone up considerably since (if the growth in the number of installations is anything to go by: 100,000 -> 600,000).

Standards

In an effort to control the rising cost of porting applications software to different operating systems (or even different flavours of the same operating system), the IEEE set up a working group to propose a portable operating system interface standard. The name chosen for this standard, *POSIX*, gives ample clues to its heritage. It is very likely that this standard will be adopted by ANSI and, ultimately, even ISO [STE88].

In the same vein, there have been a number of initiatives by computer manufacturers aimed at providing a common applications environment. This would allow users to build applications software which could be transferred, without change, to any system conforming to this standard. Both these movements (*X/OPEN* led by major European computer manufacturers and *Open Systems Foundation* led by IBM and other US computer corporations) have pledged their support for POSIX and, hence, accepted the commercial importance of UNIX.

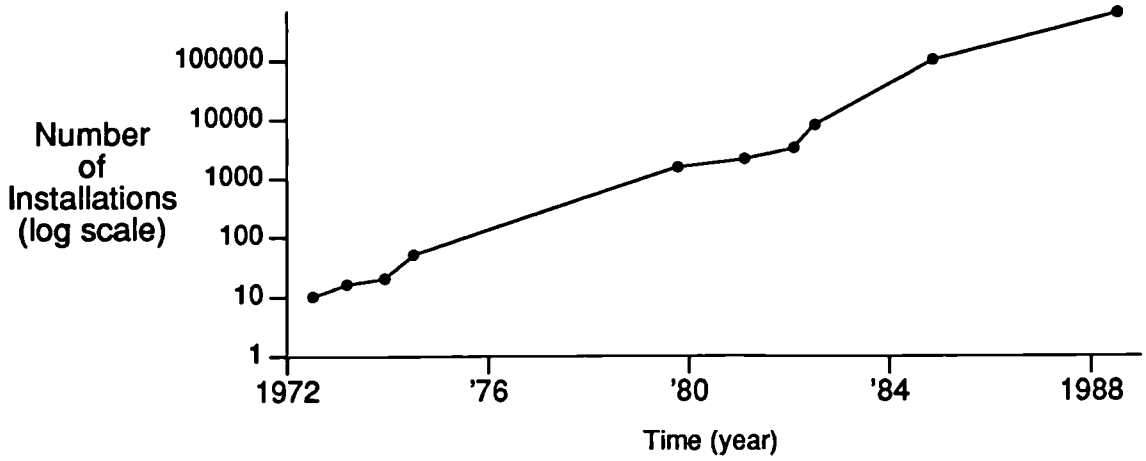
Publications and awards

The scholastic interest in UNIX can be judged by the number of publications about it. Hundreds of books and thousands of papers, from all over the world, have been published on the subject. There are bookshops and services dedicated solely to dealing with UNIX publications. There are numerous organisations whose sole objectives are to serve the UNIX community (e.g. USENIX, /usr/group, Uniform, EUUG, AUUG). The 1982 IEEE Emmanuel Piore and the 1983 ACM Turing awards were given to Ritchie and Thompson, the principal authors of UNIX, for their invention.

1.2 THE PROBLEM

Clearly, UNIX is one of the (if not *the*) most significant software products of our time. Although it has not been as successful in the business computing market as it has been in the scientific and technical community, it is bound to succeed there once the relevant applications are available, considerably expanding its user base of over half a million installations. The recent work on standards will ensure that applications flood in. In a survey by Yates Ventures [YAT82], it was estimated that commercial installations accounted for over 90% of the UNIX user base, while in 1979, they amounted to no more than 3%. The rapid growth in the popularity of UNIX can be seen in the following graph, which displays an almost exponential rise:

POPULARITY OF THE UNIX SYSTEM



Life will become very difficult for software which does not meet standards such as POSIX. Indeed, there are signs that this is already happening as the US Department of Defence (the largest consumer of software in the world) has stated that all software supplied to it must conform with POSIX. Therefore a large proportion of the computer community, and therefore, indirectly, modern society, is investing in UNIX.

Is this investment in UNIX justified?

How would we go about finding out? How would we evaluate investment in anything? Let us consider an automobile.

When evaluating a car, at a given price, we usually look at three aspects of the car:

1) Performance
2) Features
3) Structural integrity

Performance would include measures like acceleration, top speed, fuel economy, handling and road holding, ride comfort and refinement. In the second category we would look at the 'show room appeal' of the car: how roomy it is; are the seats comfortable; is driving position ergonomically sound; does it have creature comforts like air-conditioning, central locking and electric windows. Finally, we would look at its structure: what it 'feels' like; what is the paint finish like; do all the panels fit properly; does the door shut with a 'thud'; what is its reliability record.

In addition, if we are evaluating a used car we look at its service history and see how the car has responded to use and interaction with its environment. To a large extent, a car's resale value is determined by its structural integrity.

Logic dictates that the same three aspects be covered when investigating other goods. In software, we look at performance (response time, memory requirements, etc.) and features (quality of user interface, manuals etc.) but we rarely take more than a cursory glance at structural integrity. Perhaps, because we are under the illusion that software does not 'age' or 'rust'.

It is true that software does not decay spontaneously, however, research has shown that, as changes are made, the original structure of the program deteriorates [LEH74] making it increasingly difficult to understand and change. It is now widely accepted that all non-trivial software undergoes continuous change, as UNIX has demonstrated.

The current interest in UNIX means that UNIX system is about to be bombarded with a large amount of change requests. The task of this project is to evaluate whether UNIX is strong enough to withstand this onslaught. As the cost of replacing software increases, it is vital to test UNIX's ability to keep up with rapidly advancing technology and continually changing user environment.

This project aims to predict how UNIX will respond to change in the future (i.e. how structurally sound it is) by looking at how UNIX has responded to change in the past.

1.3 OBJECTIVES OF THE PROJECT

The overall objectives of the project were:

- To use a *quantitative software life cycle modelling technique* to examine the evolution of the UNIX system and attempt a prognosis for its future.
- Hence, critically evaluate the validity of the concepts used in this technique[4] and comment on the usefulness of such metric modelling exercises.

4. This technique is derived from Belady and Lehman's concepts of Program Evolution Dynamics. It claims to be able to predict the future behaviour of software systems but hasn't been sufficiently evaluated. See discussion in next chapter.

1.4 OVERVIEW OF THE THESIS

Chapter 2 discusses why the most appropriate metric modelling technique to examine the evolution of UNIX is the one suggested by Lehman. Then, the development of Lehman's concepts is reviewed with particular emphasis on the empirical evidence supporting his 'Theory of Program Evolution' and independent evaluations.

Chapter 3 relates the history of the UNIX systems by following the sequence of UNIX releases from the main centres in the Bell System[5] and the University of California at Berkeley. Thus, it introduces the releases to be modelled in Chapter 5.

Chapter 4 describes the tools and techniques used to arrive at the models presented in this thesis. This includes a discussion on which metric measures were found to be best suited for modelling the desired UNIX attributes. The configuration management strategies of the various UNIX centres and the resulting difficulties in obtaining the required information are also described.

In Chapter 5, an attempt is made to construct statistical models of the UNIX evolution process by plotting the progress of the three main branches of the UNIX evolution tree (Research UNIX, the System V stream and BSD/UNIX) in terms of changes in various system and process attributes such as size, growth-rate, work-rate and staffing. The implications of these findings on the future of UNIX and the validity of Lehman's concepts are also discussed.

Chapter 6 discusses how the dynamics of the programming process are affected by cultural differences in three programming environments (research, commercial and academic). In particular, how these differences affect the scope of "The Theory of Program Evolution".

Chapter 7 summarises the findings of this investigation and points to possible future developments.

---

5. Before its divestiture in 1984, the AT&T empire was known as the Bell System. In this thesis, for the most part, the terms *AT&T* and *The Bell System* mean the AT&T company at the time, which includes Bell Laboratories.

Chapter 2
REVIEW OF PROGRAM EVOLUTION CONCEPTS

"It is an error to imagine that evolution signifies a constant tendency to increased perfection. That process undoubtedly involves a constant remodelling of the organism in adaptation to new conditions; but it depends on the nature of those conditions whether the direction of the modifications effected shall be upward or downward."

(T. H. HUXLEY, 1888)

2.1 INTRODUCTION

Evolution is a generally accepted fact of life. While it has been recognised for some time all complex artificial systems evolve [SIM69], the treatment of evolution has been at the level of successive generations. Weinberg was the first to discuss evolution as applying to *individual* artificial systems [WIE70]. Since then, the computer community has accepted that evolution is intrinsic to computer systems in general and software in particular. Indeed various studies have shown that as much as 70-80% of the life-cycle expenditure on a program is *after* first installation [LIE80].

The previous chapter describes the primary objective of this study as forming a prognosis for the future of the UNIX system. This chapter briefly discusses some of the techniques used in the past for doing that, choses the most suitable one and reviews that technique in detail.

2.2 COST ESTIMATION TECHNIQUES

Over the years various people have suggested ways to estimate the costs and resources involved in producing software. The earlier attempts, e.g. [NEL66], concluded that there were too many nonlinear aspects of software development for a successful linear cost estimation model. Since then, researchers have been more successful [WOL74], [PUT78], [FEL77], [BOE81], [BAI81],

[JEN83], [JEN88] inter alia. Most of these (surveyed in [JEN86]) have an estimation equation of the form:

$$MM_{nom}=c(KDSI)^x$$

where

MM_{nom}	is the nominal effort required (in man-months),
c	is a constant,
$KDSI$	is thousands of delivered source instructions and
x	is an index.

Usually this equation is supplemented by a set of adjustment factors taking into account attributes such as personnel capability, application complexity and use of modern programming practices. The differences between the models are mostly in the values of the constants and indexes and different adjustment factors.

Size estimation problem

All the models which have the form given above rely on estimates of the final size of the program. This is a very difficult problem in itself but some progress has been made, in for instance [ALB79], [ITA82] and more recently in [VER87] and [JEN88].

Evolution problem

Most of the cost estimation techniques described in this section estimate the effort required to produce and first install the software, not its cost after installation.[1] However, as described in the previous section, most of the cost associated with software is after first installation. Furthermore the target of this analysis (the UNIX system) had been in widespread use for a number of years and a long term prognosis was required, particularly on how the system would respond to change. The techniques described above, and the computing community in general, do not seem to attach enough importance to maintenance [LEO88].

1. At the most, some estimate is given as to the effort required to remove the bugs present in the system at the first release [KUH82]. Other methods simply multiply the development cost by the ratio of maintenance to development cost [JEN86].

Quantitative models of software life-cycle evolution

The high rate of evolution of software (compared to biological, sociological, etc.) systems makes it possible to observe significant change in a comparatively short span of time. The studies of Belady and Lehman have shown that it may be possible to statistically model this evolution and use it to give a prognosis for a software system's future [BEL76]. Hence their technique seemed to be the most suitable one for this investigation.

2.3 A REVIEW OF PROGRAM EVOLUTION DYNAMICS

The studies of Belady and Lehman have revolved around the concept of program evolution as a dynamic process, interacting with the environments in which it exists. This, and related concepts, have culminated in a "Theory of Program Evolution" which is summarised in the five 'laws' and described in the next section. This section briefly reviews the development of these concepts, a much more detailed review can be found in [BEL85].

2.3.1 Origins

In the late 1960's Lehman undertook a study of the programming process as practised in IBM with a view towards suggesting research projects that would seek to increase IBM's programming productivity. The report, [LEH69], included a brief analysis of the "programmatic characteristics" of OS/360, IBM's first general purpose operating system. in particular, he commented on its continuing growth and increasing complexity.

The report led Lehman to ask Belady to help him model some of the reported observations. As a result [BEL71] introduced the concept of "Programming Systems Growth Dynamics" and recognised the intrinsic evolutionary nature of software.

2.3.2 The 'laws' of Program Evolution

Still based on OS/360, [BEL72] introduced more data and plots of the system's evolution (which were later to become definitive). They observed the unexpected smoothness of some plots while exponential growth in others. These concepts continued to be refined, with the emphasis now shifting towards interpretation of the data and understanding the underlying concepts. This led to the replacement of the term *growth* dynamics by *evolution*, and by the time of Lehman's inaugural lecture at Imperial College [LEH74], there was already a coherent theory of "Program Evolution Dynamics". The theory was summarised in three 'laws':

I *Law of continuing change*
A system that is used undergoes continuing change until it becomes more economical to replace it by a new or restructured system.

II *Law of increasing entropy*
The entropy of a system increases with time unless specific work is executed to maintain or reduce it.

III *Law of statistically smooth growth*
Growth trend measures of global system attributes may appear stochastic locally in time and space but are self-regulating and statistically smooth.

The paper also stressed on the importance of achieving the right balance between progressive and anti-regressive activities.

More systems were examined from an evolution dynamics (ED) point of view and the generality of the observations discussed [BEL76], [LEH76] until [LEH78] in which a detailed discussion of the state of the art in ED led to the formulation of two further laws and a cleaning up of the first three:

I *Law of continuing change*
A program that is used undergoes continuing change or becomes progressively less useful. The change process continues until it is judged more cost effective to replace the system with a recreated version.

II *Law of increasing complexity*
As a large program is continuously changed, its complexity, which reflects deteriorating structure, increases unless work is done to maintain or reduce it.

III *Law of statistically regular growth*
Measures of global project and system attributes are cyclically self-regulating with statistically determinable trends and variances.

IV *Law of invariant work rate*
The global activity rate in a large programming project is invariant.

V *Law of incremental growth limit*
For reliable, planned evolution, a large program undergoing change must be made available for regular user execution (released) at maximum intervals determined by its net growth. That is, the system develops a characteristic average increment of safe growth which, if exceeded, causes quality and usage problems, with time and cost over-runs.

Clearly, the fourth and fifth laws were particular instances of the third law. The other significant change has been the replacement of the term *entropy* by *complexity* to reflect the research being conducted into program complexity at the time. Consolidation and further refinement of these concepts continued until [LEH80] changed the phrasing of the last three 'laws' into their final (to date) form:

III *The Fundamental Law of Program Evolution*
Program evolution is subject to a dynamics which makes the programming process, and hence measures of global project and system attributes, self-regulating with statistically determinable trends and variances.

IV *Conservation of Organisational Stability (Invariant Work Rate)*
During the active life of a program the global activity rate in the associated programming project is statistically invariant.

V *Conservation of Familiarity (Perceived Complexity)*
During the active life of a program the relase content (changes, additions, deletions) of the successive releases of the evolving program is statistically invariant.

That paper discusses the nature of these laws, the consequences of violating them and illustrates

that by giving a detailed worked example.

2.4 EXPECTED BEHAVIOUR

2.4.1 The measurements

The 'laws' described above were based on observing the behaviour of various software systems in use. The measurements they (Belady and Lehman) used to chart the progress of system were, for each release of the system:

1. System (source code) size in number of modules.
2. The Release Sequence Number (RSN), giving the number in the sequence of releases from the first release under examination.
3. The Release Interval in months, weeks or days. This measures the time interval between the previous and the current release.
4. Modules handled. This counts the number of modules worked-on during the interval between the previous release and the current one.

Based on the above measurements, they calculated the following additional attributes, as described below:

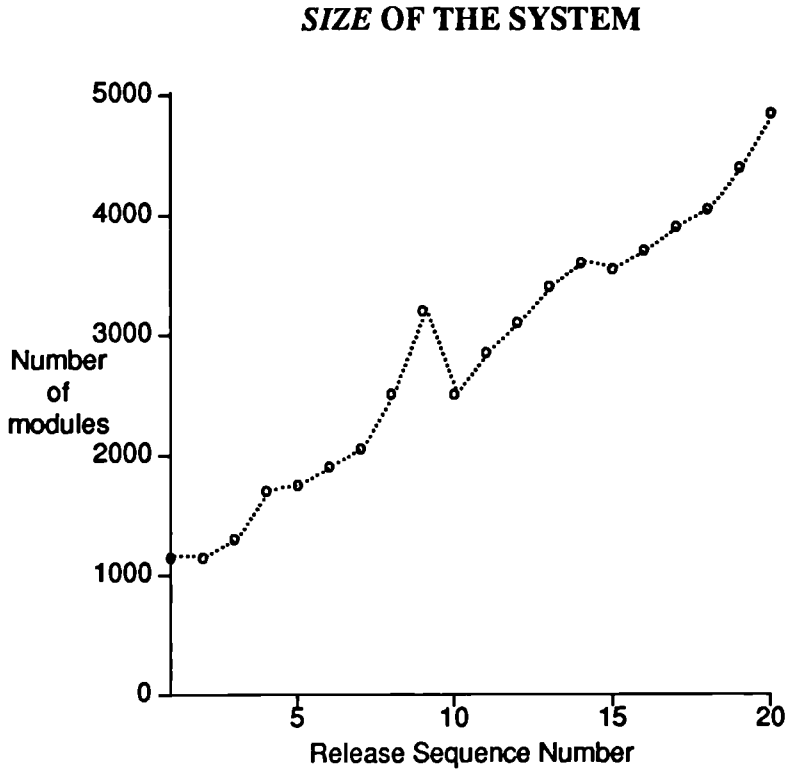
Work-rate — The average work rate during the release equal to the number of modules handled divided by the release interval.

Complexity — Defined as the fraction of the system impacted by work for the new release and measured as number the modules (of the previous release) altered during the release interval divided by the total size of the previous release.

2.4.2 The evolution pattern

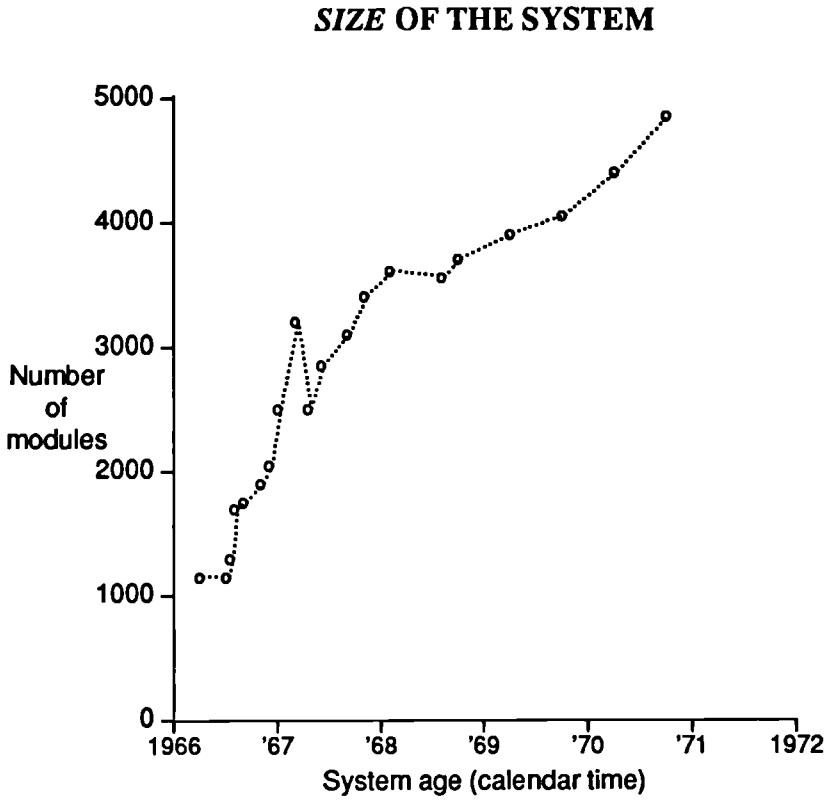
In his examination of OS/360 [LEH69], Lehman plotted the above attributes for a number of successive releases. These plots became the foundations of his "Theory of Program Evolution" and are presented below so that the reader can visualise what evolution pattern Lehman expects of a software system.

Lehman found that the system grew smoothly, with respect to release sequence number:

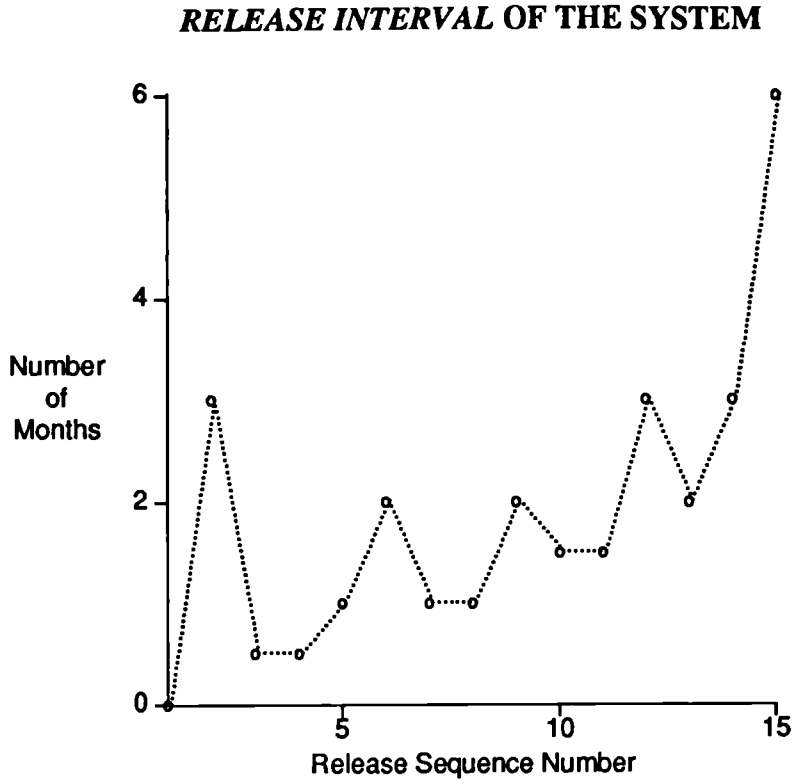


The trend indicated a linear shape with a super imposed cyclicity (of feature releases followed by clean up releases). Detailed analysis of these observations led to first and third laws.

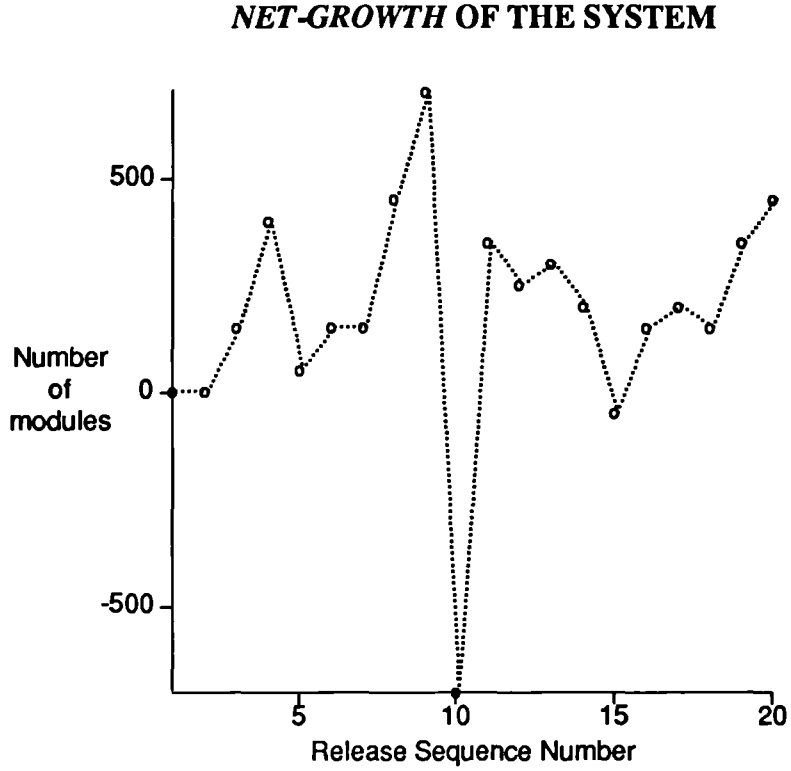
He also observed that, while systems grew, the rate of growth with respect to time was decreasing. This indicated that the system was becoming more difficult to work with.



The *Size vs. System age* plot showed an almost logarithmic growth pattern. He also observed that the release interval was steadily increasing, almost exponentially in some systems.



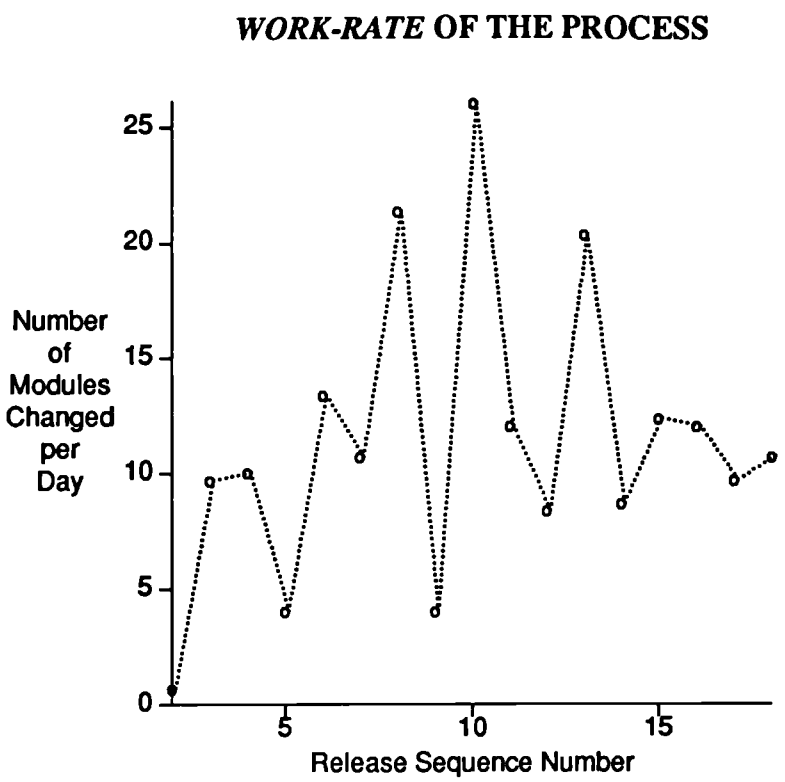
Both, the *Size vs. System age* and the *Release interval vs. RSN* plot indicated that the system was becoming more complex, if the effective amount of work done for the individual releases remained the same throughout its life-time. He duly observed that this was indeed the case:



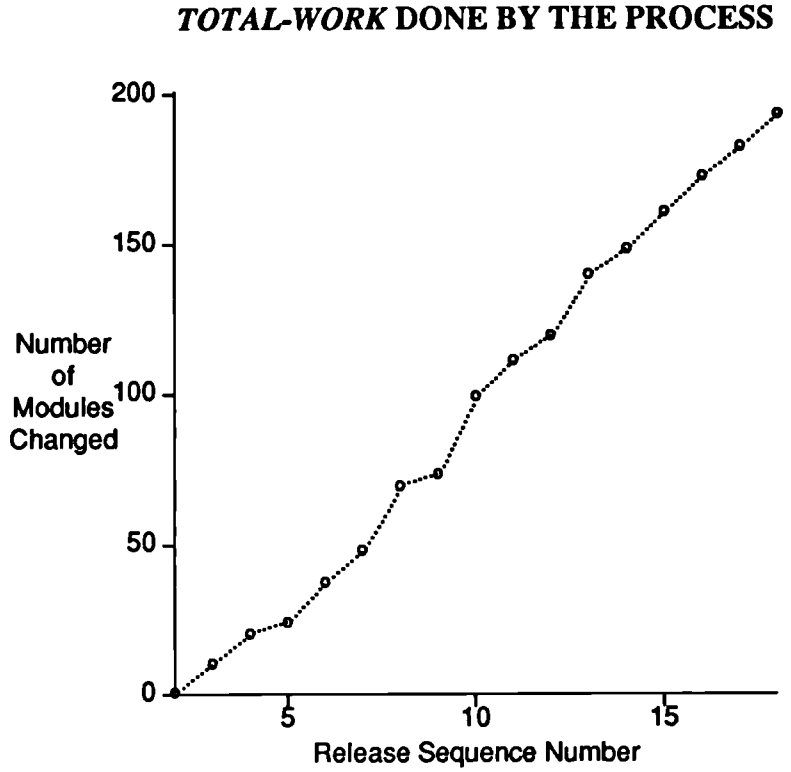
He went on to postulate that periods of very high growth were followed by periods of very low growth and hence there were limits to growth. Refinement of these ideas led to the fifth law.

In addition to increasing release interval and declining growth rates, Lehman found another measure of increasing complexity. Also, dependent on the above assertion that the effective release "content" remained constant, he suggested that if more of the previous system was being changed to accomodate an equivalent amount of new work, then the system was becoming more structurally intermingled and hence more difficult to work with. He saw a steady rise in this figure for some systems.

He also observed that, while work-rate fluctuated between individual releases, in the long term it remained constant with an identifiable mean. He elaborated that by "constant" he meant that sharp peaks above this mean value would be followed by sharp troughs below this mean value:



This led to the fourth law and was further reinforced when he saw that the total work done by the process (an accumulation of work done in individual releases) showed an almost linear rise:



Since examining OS/360, a number of other systems have been investigated with a view towards extending the "Theory of Program Evolution" and whilst a number of similar results have been found, there are also some conflicting ones. These investigations are described in a later part of this chapter.

2.5 THE SPE CLASSIFICATION AND OTHER IDEAS

Based on the insights gained in the ED studies, particularly the perception that evolution is intrinsic to the very nature of programs, Lehman [LEH80] introduced a program classification scheme, called *SPE*, which isolated the sources of evolution. Throughout these studies, the ED concepts have always been associated with program *largeness*, the SPE classification was an attempt to improve on the previous classification of large and non-large programs. In this scheme programs are classified as follows:

S S-programs are programs whose function is formally defined by, and derivable from, a specification.

P These are programs which attempt to solve a problem which can be precisely formulated but whose solution necessarily reflects an approximation of the real

world.

E They are programs that mechanise a human or societal activity and that are, therefore, embedded in the application environment in which they exist.

While S-type programs may be proved correct (since their *only* success criterion is equivalence to the specification), P- and E-type programs will continue to be changed after installation since they mechanise some part of the real world and the real world is continuously changing.

Programming Support Environments

The ED studies identified two sources of program evolution:

(1) limitations of current programming technology, and

(2) changes in the application environment.

While advances in technology (e.g. program transformation, runnable specifications, 4GLs) will help reduce the need for changes arising out of the first category, the real world is continuously changing and, therefore, programs will *have* to be changed to keep pace.

This realisation, that evolution is intrinsic to the very nature of programs, led to the search for tools and techniques to support this continuous change and eventually led to the concept of programming support environments [LEH82]. In the same vein, [LEH84] introduced the LST model of program development that provided a canonical step paradigm that can form the basis for a coherent programming process.

Mathematical Models

In the late 1970s, two researchers, Professors Riordan and Woodside, from Carleton University in Ottawa, Canada, with an interest in control theory studied the ideas of ED with that viewpoint. Their publications, [RIO77] and [WOO79] came up with simple mathematical models for the evolutionary growth of software and concluded that alternating between feature and cleanup releases would be a better strategy than one with favours smooth growth.

Recently, Daglish [DAG88] described a function of the form

$$y=k_0+k_1t+k_2e^{-at}$$

to fit some system evolution data of VME/B (taken from [KIT82]). He showed that the fit was better than a straight linear or straight exponential fit taken separately.

2.6 EMPIRICAL EVIDENCE

The 'laws' described in the previous section are based on common phenomena observed in number of systems of dissimilar size, structure and development organization.

Primarily, IBM's OS/360 operating system was studied [LEH69], [BEL72] and [BEL76]. The second system to be studied from an ED point of view was also an IBM operating system: DOS [LIM75]. Shortly thereafter, Midland Bank's EXECUTIVE system, for scheduling the day's batch processing work, was examined in [HOO75] with more data in [CHO77] and [BEN79]. The results of modelling UNIVAC's OMEGA transaction operating system were reported in [LEH76]. Centrefile Ltd.'s system for servicing Building Societies, BD, was examined in [CHO77] with more data in [CHO81]. Preliminary results of studying ALMSA's CCSS, a major military stock control system, were published in [LEH77] with more analysis in [HOG78]. Finally, the amount of development data available for ICL's VME/B (later VME/2900) operating system prompted no less than three independent studies: [CLA78], [CHO81] and [KIT82].

Some of the statistics of the systems listed above are summarized in the table below:

System name	system type	org. type	anal. period	size	releases	users
OS/360	Operating system	Computer man.	1966-1975	1152-5300	21	v. many
DOS	Operating system	Computer man.	8 yrs	438-2142	31	v. many
EXECUTIVE	Banking system	Financial inst.	1973-1978	657-967	12	2 sites
OMEGA	Transaction OS	Computer man.	1972-1974	335-388	9	many
BD	Applications	Financial inst.	1973-1977	42-66	not release based	few
CCSS	Stock control	Software hse.	1972-1979	971-1483	58	few
VME/B	Operating system	Computer man.	1975-1982	1728-3239	10	many

where the sizes are given in number of modules.

2.7 CRITICAL EVALUATIONS

In a critical analysis of ED work, Chong [CHO80] concluded that there was little evidence supporting the notion of "statistically smooth" growth. He preferred to say the patterns showed "no time dependent behaviour" and that most of the observations could be explained by viewing them as Normal Distributions.

Indeed, in another study, [LAW82], laws 3-5 did not stand up to rigorous statistical scrutiny and hence software managers are more in control of their projects than Lehman would have us believe. However, Lawrence did find some evidence supporting laws 1 and 2.

On the other hand, the most recent study [KIT82], found most of the results in accordance with those predicted by Belady and Lehman. However, she found no evidence of the characteristics associated with increasing complexity in VME/B.

2.8 NEEDS

Boehm [BOE84] commented that some types of systems seem to obey the 'laws' more than others. Therefore, he and other researchers [CON86] recognised further work in this field as an important area of research. Furthermore, all the critical studies, cited above, admited that their evaluations would benefit from more data and better measures, particularly in the area of process attributes.

There was a need to identify what characteristics of those systems and processes make them more amenable to statistical modelling. The SPE classification scheme redefined the scope of the proposed theory with respect to *programs* but no such scheme existed for *processes* or environments.

This study sought to extend the theory in this area by examining three UNIX systems, evolving in very different environments. Thus the ED database was extended and an attempt was made to identify the effect of the process environment on the dynamics of the system.

UNIX was also a good system to study because it was well known for its quality of implementation and had been largely under the control of its implementors (free from management hindrance). It would have been interesting to see if it also suffered from the phenomena predicted by Lehman.

Chapter 3
THE HISTORY OF UNIX

"The number of UNIX installations has grown to 10, with more expected."
(The UNIX Programmer's Manual, 2nd Edition, June 1972)

3.1 INTRODUCTION

UNIX has grown in stature from being Ken Thompson's research toy to becoming one of the most influential software products of our time [MAR84]. This chapter attempts to capture on paper this fascinating piece of history by presenting the important milestones in the life of the UNIX system. The focus is on the UNIX releases from the major centres of its development in the Bell System and the University of California at Berkeley, both in the United States.

The purpose of this chapter is firstly, to introduce and describe the releases which are used to model the UNIX evolution process in the next chapter so that the reader is not lost in the confusingly similar release numbering schemes adopted by the UNIX groups under study. The chapter will also enable the reader to see if the models pick up any of the several organisational changes affecting the evolution of the system. The second major aim of this chapter to is present a definitive account of the system's history. Several versions of its history have appeared in the literature, most notably [RIT78], [RIT79], [DAR84], [FEL84] and [MCK85], but they have mostly concentrated on the technical developments in the research versions and there was a need for a comprehensive history of its pre--commercial era, especially on its non-research versions. Furthermore, the previous historians did not have access to the sources that were (eventually) made available to this study (for extracting numerical information for model construction).

Structure of this chapter

After a brief account of the origins of the system, the chapter traces the history of UNIX by following all its major releases. The releases and organisational development of each Bell System

centre are discussed in turn. Finally some of the background and highlights in the licensing of UNIX is also given.

Sources

The lack of proper record keeping by *all* the UNIX groups has resulted in a large amount of valuable "hard" information being lost, so this study (and, to a greater extent, others before it) has had to rely on informal records and recollections of pertinent staff.
This section has been pieced together from almost complete collections of the *UNIX Newsletter* (first published by the USG, see below), *;login:* (the newsletter of USENIX, see below), licensing records, System Release Descriptions, Manuals and even (for organization charts) old Bell Labs telephone books!

3.2 ORIGINS

UNIX might never have come into being if Bell Labs management had not deemed the MULTICS project a failure and pulled out. This section traces the origins of the system: from the withdrawal of Bell Labs from the MULTICS project to the publication of the first UNIX Programmer's Manual (i.e. its first release). It concentrates on the motivating factors for the formation of the system.

Bell Labs' interest in operating systems goes back to the 1950's when the rapid increase in the operating speed of the computers meant that some of the tasks that operators performed for the early relay machines and their successors had to be automated. Such programs (initially called monitors) were not common place and when the Labs acquired faster machines they had to write their own. The first released one was *BESYS-2*, launched in April 1958 for the IBM 704 machine [HOL82].

Sucessive versions of BESYS systems continued to serve Bell Labs until *CTSS* (for Compatible Time Sharing Service) was introduced by M.I.T. [CAR62]. Shortly afterwards, in 1964, Bell Labs joined forces with M.I.T. and General Electric to develop a successor to CTSS called MULTICS (for MULTiplexed Information and Computing Service) [CAR65]. It was an ambitious system intended to provide access to a central GE 645 computer for a large user community via separate remote consoles [ORG72].

By the late 60s it was obvious to Bell management that a single central computer complex would not be able to meet the diverse needs of a large research and development organization, MULTICS had failed to deliver. Also the prototype version of the system was proving very expensive to run. In addition there were difficulties in coordinating the development effort between the researchers at Bell Labs and their counterparts at M.I.T. Finally, in 1968, Bell Labs

decided to pull out of the project.

The withdrawal of Bell Labs from the MULTICS project was particularly disturbing to the last technical staff to be involved in it. Although exorbitantly expensive, the system did provide a service (at least to them) that was friendly. In 1969 they started looking for ways to recreate the pleasantly interactive computer service that MULTICS had offered them. Not finding a suitable alternative, they started to design their own system. Since the GE 645 (the machine they used to simulate MULTICS and test their ideas) was soon to go away, they also tried to obtain finance to purchase a suitable machine.

In the meantime (still in 1969), Ken Thompson (a researcher in the Computing Science Research Center {CSRC} at Bell Labs, Murray Hill - one of the last to withdraw from MULTICS), while converting a simulation game that he had written for the GE to a dis-used PDP-7 computer,[1] discovered that programming on the PDP-7 was a difficult task indeed. Since their original quest for machine resources was rejected by the mangement,[2] the researchers had to make do with the PDP-7. They soon decided that it would much easier in the long term if they rewrote the system software for the PDP-7 themselves, incorporating their versions of the good ideas arising in the aftermath of MULTICS. Thompson, with the help of others, set about doing this and soon the system was able to support itself. As a pun on MULTICS, it was called UNIX.

Since they could not pretend to offer a computing service without *Fortran*, Ken Thompson set about writing a compiler for it but he ended up writing a definition for a new language based on BCPL. Dennis Ritchie then took over development of this language (now called 'B' from BCPL) which ended up, as described below, as the popular 'C' programming language. Although 'C' replaced Fortran as the staple high level programming language at the CSRC, interest in Fortran did continue, particularly by Stu Feldman who, along with Peter Weinberger, wrote probably the first complete Fortran 77 system, **f77** [FEL78].

The research team made another attempt at obtaining a better machine (a PDP-11), however, this time they specifically promised to deliver a text processing system for the patent office to prepare their applications on. The proposal was accepted and the machine duly arrived. B was ported to it

1. The game was very expensive to run on the central GE machine and since the PDP-7 was a 'spare' machine, he would total control over it (hence it would be much cheaper). Furthermore, the PDP-7 had a very good display processor.
2. The management did not want to support further operating system research so soon after the painful withdrawal from the MULTICS debacle.

almost immediately but did not suit the architecture of the new machine so work began on another language 'C'.

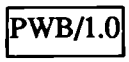
By the summer of 1971, the text processing had been implemented and patent office was successfully preparing their patent applications on UNIX, hence fulfilling their charter.

3.3 THE UNIX RELEASE TREE

In the early 70's senior management at Bell Labs sought to rationalise the purchases of computer hardware in the Bell System so they set up a group to oversee these purchases. This coincided with the popularity of mini-computers in general and the DEC PDP-11 in particular. Impressed by the in-house developed UNIX system, the person in charge of the purchasing group (Berkely Tague, later to become head of USG, see below) twisted the arms of all the departments with requests for the PDP-11 to accept UNIX as the operating system. Not only was UNIX (in his eyes) a superior product but it would help to make Bell less dependent on the vendors.

Since UNIX was initially provided without support, several groups had to, and did, develop their own expertise and tailored the system to meet their own needs. This phenomenon repeated itself, after a time lag of a few years, outside the Bell System. Amongst the numerous groups and companies that developed (even sold) their own versions of UNIX, the histories of only the strongest (in terms of users, influence and "closeness" to the mainstream UNIX) are presented below. A summary of their inter-relationships is given in the accompanying diagram, where the releases:

are described in the **RESEARCH** section

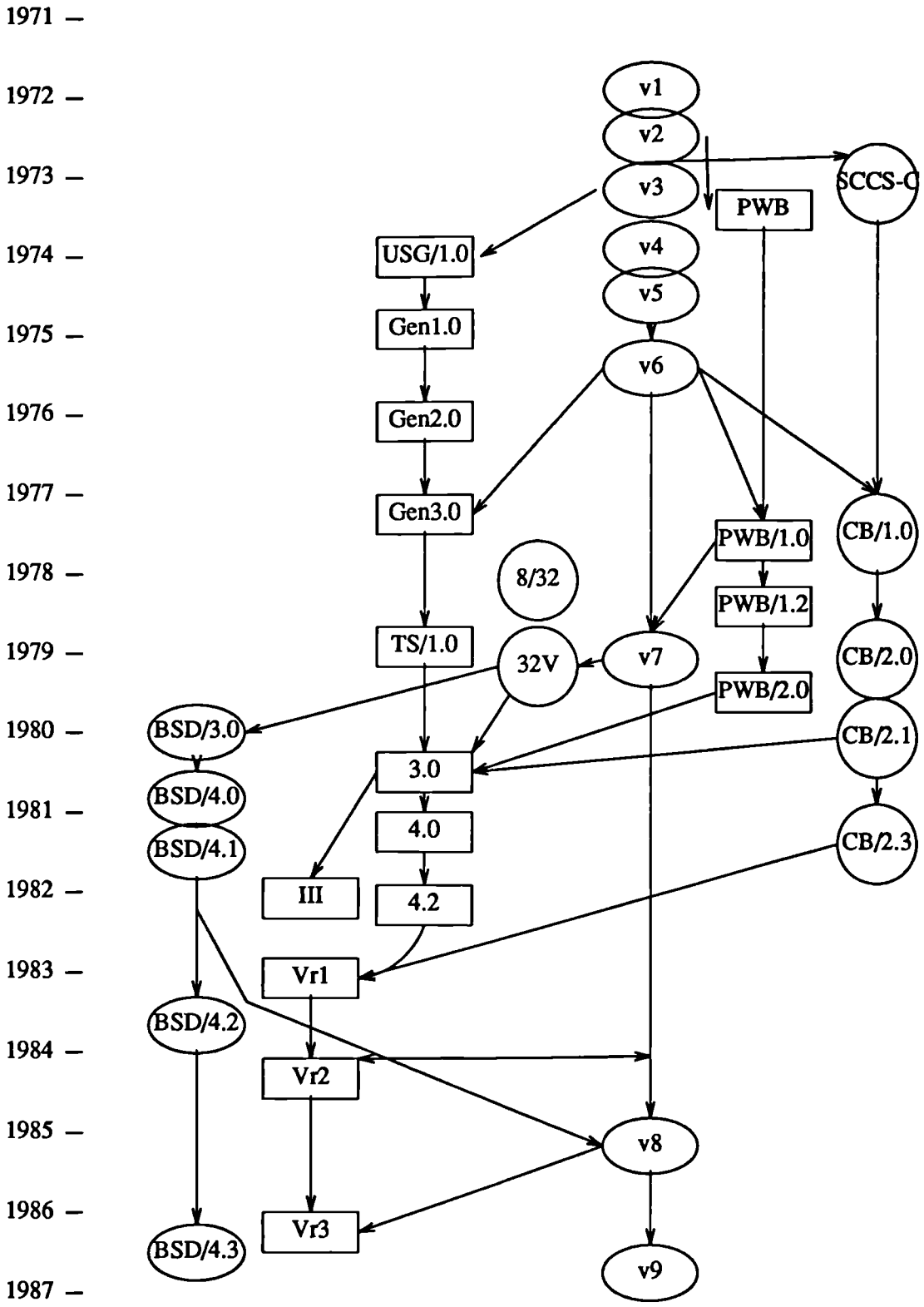


are described in the **PROGRAMMER'S WORKBENCH** section

are described in the **COLUMBUS OPERATIONS SYSTEMS GROUP** section

are described in the **UNIX SUPPORT GROUP** section

are described in the **UC BERKELEY** section



THE UNIX EVOLUTION TREE

3.3.1 Research

Ken Thompson and Dennis Ritchie published the first UNIX Programmers Manual in November 1971 describing the system as it was at the end of the *Origins* section described above. With the C compiler working, more people were attracted to the UNIX programming environment. The system improved steadily until, by June 1972, there were enough changes to warrant a re-publication of the manual. The assembler and the loader had undergone some reorganisation but the most important change in v2 was the introduction of the interprocess communication mechanism: pipe. Pipes were to become one the distinguishing features of UNIX. By this time the number of UNIX installations had grown to ten.

In the early days people who wanted UNIX systems just went along to the Research Center and took a magnetic snapshot of what was on Thompson's system at the time, there wasn't an official release or a distribution tape as such. The version of UNIX on the Research Center was known after the edition number of the current manual. So, for example, between the publication of the first edition of the manual and before the second, the system was known as version 1 or v1.

The third edition of the manual was published in February 1973, to bring it in line with the software changes to accommodate the upgrade to the PDP-11/45. A number of *applications* programs were re-written in C as the UNIX programmer population grew slightly. As the popularity of UNIX increased (the number of installations increasing to 16), more attention was paid to help the inexperienced users.

Two very significant events took place during the period leading to the publication of the fourth edition. A phototypesetter was obtained, significantly improving the document preparation facilities, i.e. **troff** [OSS77], provided by UNIX (probably its most popular use). More importantly, the complete UNIX *kernel* was rewritten in C. The system software also changed substantially to introduce multiprogramming and the ability for several user programs to share reentrant code. The fourth edition of the manual (published in November 1973) was the first to be typeset and described only the C version of UNIX.

In an article for the July/August 1973 Bell Laboratories Record, Sam Morgan (then Director of the Research Center) discussed the usefulness of UNIX, making probably the first mention of UNIX in a published paper [MOR73]. Also in this time frame, UNIX was announced to the outside world by Ritchie and Thompson in a paper presented at an Operating Systems Symposium in October 1973. A revised version of that paper was printed in the CACM of July 1974 [RIT74]. This, understandably, led to a rush in orders for the UNIX systems. The requests came mostly from universities but there were commercial inquiries as well.

The increased popularity of UNIX brought with it more contributors to the system software (i.e. from outside the Research Center) and the manual had to be republished in June 1974. Most of the work was at the user level resulting in utilities like **yacc** [JOH75], **diff** [HUN76], and **grep**. It is rumoured that some 5th Edition tapes (the first time there was an official tape) were sent out with copyright notices in the source (which was, right from the beginning, provided on-line), as instructed by the legal department, presumably to prevent un-authorised use. However, the legal department soon changed their line to say that having copyright notices implied that the software had been published and instructed the programmers to take them out. Some outside users had to be bribed with version 6 tapes to get the copyrighted 5th edition tapes back!

The outside demand for UNIX had increased so much that the then current philosophy of copying the CSRC system onto tape and then sending it to the licensee was too much for the Research Center to handle and a more formal mechanism was set up. The technical library at Murray Hill was commissioned to take care of the distribution of UNIX. The librarians were to handle all inquiries and shipments, indeed, henceforth, the Research Center had nothing to do with it, they would simply provide the tape and associated documentation to the librarian. Version 6 was the first version of UNIX to be released in this manner. The manual was dated May 1975 and documented the mostly minor changes to the system (including the release of **bc** and **eqn** [CHE75]).

The next manual (the seventh edition) was published in January 1979 to go with the tape prepared in late 1978. This was the first manual not edited by Ritchie and Thompson.[3] Several significant developments took place in this long time interval. A concerted effort was made to clean up the code and remove any PDP-11 peculiarities. In a research exercise, the whole UNIX system was ported to an architecturally very different machine (the Interdata 8/32) [JOH78]. Prior to that a portable C compiler was written [JOH78a], which was an offshoot of an M.Sc project [SNY74]. A new new file system was implemented to facilitate much large files. At the applications level, there was a new shell (command interpreter [BOU78]); **make** [FEL79], **sed** [MCM78] and **lex** [LES75] (amongst a host of others) were included in the release for the first time. Numerous improvements were made to the document processing software and to the language processors.

The advent of the DEC VAX-11/780 at the time prompted another research group at the Labs (Interactive Computer Systems) to try and port UNIX to the VAX. A couple of members of that

---

3. Doug McIlroy and Brian Kernighan taking over that role.

group (Tom London and John Reiser) managed to complete the transfer of the system in the space of six months (the version 7 look-alike first ran successfully in May 1978) attesting to the basic quality and clarity of the original system [LON78]. This system was licensed to the outside world (in February 1979) as 32/V (after the 32 bit processor) and, in spite of some problems, became the instant choice for PDP-11/UNIX users upgrading to a more powerful machine.

Another long hiatus followed before the 8th Edition came out in February 1985, although **awk** [AHO78] and device independent **troff** had been released prior to that. It was based on BSD 4.1[4] and System V (as well as version 7, of course) and ran on the VAX processor. Behind the scenes, a lot of Research Center effort was put into the development of the **Datakit** [FRA79] virtual circuit switch. A new bit mapped display terminal [PIK84] was also developed at the Research Center and version 8 included a lot of specialised software making use of these capabilities. Other v8 highlights were streams [Ritchie 1984], advanced networking and remote file systems.

Staff at the Research Center have continued to work at an accelerated pace, mostly consolidating their version 8 efforts. The ninth edition of the manual was published in September 1986 reflecting the strength of activity. There isn't a distribution tape as such, to go with the manual. In a sense they seem to have returned to the pre--v5 days when the system was evolving rapidly (but with few revolutionary changes) and the research centre staff did not bother to produce official distribution tapes.

3.3.2 Unix Support Group, UNIX Development Laboratory & AT&T-IS

In the early 1970s there was a large increase in the popularity of departmental mini systems throughout the Bell System catalysed by the introduction of the PDP-11. Soon word spread that UNIX was a preferable alternative to the supplied DEC operating system.

Some operating telephone companies and the switching control center system (SCCS)[5] group in Holmdel, NJ decided to use UNIX to collect maintenance data from their switches and for administration purposes. Other departments also started building applications on top of UNIX, some part of turnkey systems licensed by Western Electric (WECo).[6]

---

4. **BSD stands for Berkeley Software Distribution, the UNIX releases from the University of California at Berkeley are BSD/UNIX.**
5. **Not to confused with the source code control system, also called SCCS.**
6. **Western Electric was the commercial/licensing arm of AT&T. All products produced by Bell Labs were sold or licensed as Western Electric products.**

Since the Research Center offered UNIX on an as-is basis, the users had to provide their own support. Also Thompson and Ritchie were getting fed up with stupid inquiries about UNIX. By late 1972 it was clear that UNIX was here to stay and that some form of centralised support was needed to aid the various departments using UNIX. The switching control center (a very rich and influential department) decided to fund a small team and in September 1973 the UNIX Support Group (USG) was formed reporting to Berkely Tague.[7] It had one supervisor (J. F. Maranzano) and two technical staff along with some additional support.

USG's objectives were

- maintenance of a standard version of UNIX
- in-house distribution to Bell Labs projects
- documentation of system internals
- controlled further development in response to project needs

At about this time UNIX was re-written in C, dramatically increasing its popularity within the Bell System because the technical staff were more confident at dealing with a high level language than assembler. UNIX was changing very rapidly and USG's first task was to stabilize the system and act as a filter between the end projects and the gurus at the Research Center. To do this the technical staff at USG spent a lot of time with Thompson and Ritchie getting to know the system, frequently they would *start* their sessions at 5 p.m., to get their full attention. This illustrates some of the idiosyncrasies of the UNIX gurus: they come in to work at lunch time, work until 7ish (the most productive period of that being when most of the others have gone home), go home and then do the real work at home (all have terminals etc. at home) between midnight and 4 a.m.!

It took them a while to be confident enough to make modifications of their own, however, they were more successful in other areas. They initiated a semi-formal trouble reporting system along with a UNIX newsletter. Their UNIX and C courses (the first of their kind) and user group meetings were very well attended.

The first UNIX release from USG, Release 1.0, was dated 15th December 1973. The first issue of the monthly *UNIX Newsletter* was published in January 1974 and was edited by Joe Maranzano.

---

7. Head of the computer planning department (Dep. 8234) at Murray Hill.

The problems or suggestions reported to USG were published in a "trouble reports" section in the newsletter as were announcements of new products and documents. This format was continued in later issues. If USG was going to (or had taken) any action on the trouble reports, this was also reported in the newsletter.

April saw the second release from USG. At this stage USG was still mostly acting as a buffer between the customers and the Research Center. In November, UNIX became a Western Electric product (since it was being shipped out underneath some applications). It was released as issue 1 of UNIX Operating System Generic PG-1C300. UNIX releases were now called generics, and generic 1 was equivalent to USG release 2 mod level 2.24.[8]

By 1975 a small real time version of UNIX had been developed called MERT [LYC78], for Multi Environment Real Time. The UNIX newsletter was asked to cover the evolution and problem reporting of MERT as well as BOS (for Bell Operating System), a real time executive system[9] developed elsewhere in the Bell System. Reflecting these inclusions the newsletter's name was changed to *Minisystems Newsletter* but was still published by USG. By the summer of that year, support for UNIX was formalised, customers could remotely login to the USG computer and enter an on-line trouble report or alternatively contact a member of USG. To provide additional UNIX support for the Operating Telephone Companies, Western Electric had to set up an organization in North Illinois headed by Doug Green.

WECo also committed to bolster USG's manpower by loaning them two members of staff, taking the team size to 12. In August, a departmental task force was commissioned to rationalise the future of UNIX, BOS and MERT in the department. Coincidentally, the first MERT manual was announced in October.

Make (a program for maintaining other programs) was launched at the CSRC towards the end of the year and was immediately adopted by USG for the next generic release (PG-1C300 issue 2). This was a snapshot of the USG system at mod level 3.33 (January 1976) indicating at least three distinct levels of evolution: the generic releases, major and minor USG mod levels.

---

8. **USG also shipped modification packages to the customers between releases, the mod level corresponded to that.**

9. **An airline reservation type system for accessing a database from many terminals.**

In February 1976, USG decided to offer supported versions of two PWB/UNIX products to its customers (still within the Bell System) which were to become 'standard' UNIX products and contribute greatly to the success of the system: SCCS and the MM macro package (they are described in the PWB section). In addition, RJE was also offered but without support.

Generic 3.0 was released in spring 1977 (delayed from January). It contained new drivers, communications software and nroff/troff. It also included SCCS, MM and sed.

In June, a task force (called the MOPS committee) set up to standardise the use of minicomputer operating systems in the Bell System recommended that PWB and USG UNIX should be combined into UNIX/TS (for Time Shared), which would be based on v7 and would include CB/UNIX[10] features. The real time development of UNIX would be carried on in UNIX/RT, to be based on MERT and would look like UNIX/TS. In response to this, it was decided that PWB/UNIX would be supported by both USG and PWB group, consequently PWB/UNIX section was added to the newsletter in July. In September, it was announced that UNIX/TS would be a BIS[11] product. The base operating system would come from USG and the PWB group would add some local features (SCCS, RJE, MM) and release it. As part of a re-organization, some PWB staff joined USG.

As a result of a user meeting in March, a task force was commissioned to determine a standard shell.[12] Dale DeJager (from CB/UNIX) proposed a shared memory scheme which was also put under investigation.

UNIX became an official Bell Laboratories trade mark in November 1977. A month later MERT Release 0 (PG-1C600) was announced.

The February 1978 UNIX/MERT users meeting endorsed the MOPS committee recommendations and it was decided that generic 3 would continue to be supported but the level of support would be reduced. In March Ted Dolotta took over USG, while Maranzano headed a new group formed to collect the long term requirements for UNIX and C.

---

10. The UNIX releases from the Columbus, Ohio Operations Systems Group, see below.

11. Business Information Systems (Area 90 in the Bell System).

12. At the time Bourne (at the Research Center) had released his new shell. J Mashey (in PWB group) had his own shell and there was the original (Thompson) shell.

UNIX/TS Edition 1.0 was released in November 1978, as a successor to PWB 1.2. The main features were the new file system, user & group ids, shell. The utilities **awk** and **tar** were also included. Most of these features were imported from v7. Subsequently, a small update release (TS 1.1) was announced in February 1979. Its corresponding manual, however, was not available until August. The first edition of UNIX/RT was released in April 1979 replacing MERT. USG officially entered the VAX era with the release of TS 1.2 in September. A minor update to RT (making it 1.1) was also issued in September.

The importance of UNIX grew in the company, so much so that when AT&T decided to enter the hardware business, the first machine was specifically designed to be a UNIX engine. This was to become the 3b range of computers. A major organization change in early 1980 brought the PWB and USG groups together under the "Microsystems and UNIX Development Laboratory". In the new laboratory, Berkely Tague remained responsible for DEC operating system development while Andy Hall (who is currently a director in AT&T Information Systems, the current owners of UNIX) headed a department taking care of system testing, integration and support. At this stage the UNIX staff in the lab. numbered well over 100.

June 1980 saw the introduction of UNIX Release 3.0, the first release from the new laboratory. 3.0 replaced UNIX/TS 1.3 and PWB/UNIX 2.1. It incorporated line disciplines and 32 bit signals from CB/UNIX and included a virtual protocol machine. Improvements to the C library and other bug fixes were also included. Some minor problems were fixed by an update release (3.0.1) in September. A Bell Laboratories -wide reorganisation in January 1981 resulted in the UNIX Lab. being renumbered. Release 4.0 was launched from this organization in March. It introduced new IPC mechanisms[13] as well as new drivers and changes to the text processing software and the C libraries etc. Also in March, the first UNIX release for an IBM machine was launched as UNIX/370 Release 3.0 taking the UNIX hardware performance range to 50:1.

In August 1981, UNIX (denoted: 4.1.1[14]) was released for the WECo 3B-20s processor. It was meant *only* for the 3B machine and was basically 4.0 with hardware related changes. This release also marked the point where WECo became the official UNIX release agent (taking over from Bell Labs). An update (4.1.2) was released in December containing some memory management

---

12. Actually, due to an oversight, the supposed fixes weren't sent on the tape. Instead, the same version of the sources (the ones sent with 3.0) were on the tape!!

13. Shared memory, messages, semaphores and process locking.

14. Release 4.1 never making it out of the door as it was not meant for floating point hardware.

fixes and added on-line diagnostics.

System III was released outside the Bell System in January 1982. It was based on Release 3.0.1 and was the first time that a USG based system was licensed outside the Bell System.[15]

Release 4.2 was launched in February 1982 for both the 3B & the DEC machines. It contained improvements to the data communications and networking software and more mature IPC. Due to insufficient testing this release was labelled provisional, pending the release of 5.0, scheduled for October. Before that, however, a minor update package (4.2.1) was released in May, to fix some urgent problems.

Much improved performance, a new file system, new **init** and **getty** (from CB/UNIX) and networking with other bits from BSD/UNIX were the main features of Release 5.0, announced in October 1982.

Until then, U.S. anti-trust laws implied that since AT&T (i.e. the Bell System) was a public service company, offering a telephone service to the public, it could not *sell* software, since that was not its main line of business. Hence software was simply *licensed* to outside users, not sold on commercial terms. However, AT&T's divestiture in 1983 allowed UNIX to compete in the commercial market place and on 1st January 1983, Commercial System V was announced. As it was identical to Release 5.0, this was the first time that current version of UNIX in the Labs was licensed outside.

In September 1983 an update package (Sys. V release 1.1 corresponding to internal 5.0.5) was shipped to fix some problems with **UUCP** and **f77**. A change in the US Export Laws prohibited exporting crypting algorithms outside the US and Canada so an International System V (Release 1.0) was launched in January 1984. It was identical to System V except for the **crypt** utilities.

To allow System V to be efficiently tailored to (by now commercial) external customers' needs, some unbundling[16] took place in Release 2.0 of System V in April 1984. This effected on-line documentation and some networking code. This release featured improved performance (particularly in the shell) and new job control (in the form of shell layers).

---

15. Some developers complained that since System III was lagging behind the internal release by one year, it did not reflect Bell progress in UNIX (an outside release based on 4.0/4.2 would) but were over-ruled.

16. A technique used to remove code from the next release which has been in previous releases. Changes are made to the system to prevent customers from installing old versions of the code on the new release.

In summer, an update package was released by AT&T Technologies which fixed some problems in System V Release 1.1 (hence called System V Release 1.2). In August, paging was finally introduced in an official AT&T Release. This feature, along with improvements to the Fortran compiler and record and file locking, was released in System V Release 2.0 version 2.

In 1985 the ownership of UNIX was transferred to AT&T Information Systems (ATTIS). The staff in the UNIX Laboratory joined the Computer Systems Software Division (directed by Bill O'Shea). The development of UNIX now takes place at their Summit, NJ facility. Release 3.0 (of System V) came out in March 1986 and featured **streams** and other enhanced networking capabilities along with the DMD 5620 windowing capabilities developed at the Research Center. ATTIS has stated that it is no longer interested in supporting UNIX for DEC processors.

3.3.3 Programmer's Workbench

Frequent hardware changes in some Operations Systems departments in the Bell Systems in the early 1970s left the programming staff in the Business Information Systems (BIS) area slightly unsettled as they had to learn a new system every six months and as soon as they got fluent in it, they were changed to another system.

Even Ivie, a department head at BIS, along with other senior staff investigated a way that would allow the large development team to use a common front end to do their programming and control the sources, send it to their target machine for compilation and then see the results on the front end. These ideas can seen as forerunners of the modern Integrated Project Support Environment concept. Having decided that a mini- would be an ideal front end, Dick Haight, a supervisor there, contacted DEC who had just launched their PDP-11.

While Haight waited for the machine to be delivered, at DEC's prompt, he got in touch with Ken Thompson who had one of the first PDP-11/20s running. Thompson allowed him the use of the Research Center machine but only at night since the typists from the patents department were using it in the daytime, and required a reliable machine. He and the others at BIS were so impressed by UNIX that they immediately got a version of UNIX for their machine as soon as it was delivered and started modifying it to fit their needs, they called their system Programmer's Workbench or PWB/UNIX for short.

The development of PWB/UNIX accelerated with the C re-write of UNIX in 1973.[17] During the

next year or so, several tools were written facilitated their aims to provide a very large uniform programming environment for programs intended for multiple target machines. A facility to remotely submit jobs to the target machine (RJE) was written as well as tools for testing software and simulating the target machine [DOL76]. Further tools were written for controlling source code (SCCS - [ROC75]) and managing modification requests [KNU76].

Since PWB/UNIX was supporting a very large user community, and most other UNIX installations were much smaller, several performance improvements were made. A new disk driver was written and the process scheduling was changed as was the handling of logins and user/group ids.

By 1975, the PWB/UNIX installations were heavily used and healthy feedback from users prompted several functional changes to the shell command language and further performance improvements. The splendid text processing facilities offered by UNIX resulted in it being very heavily used for text preparation (probably more than for programming!) and this lead to a new macro package for the troff and nroff programs.

PWB/UNIX continued to evolve rapidly but the group kept up with their policy of keeping up with Research Center UNIX, changing as little as possible and trying not to spoil the underlying simplicity and elegance of the system. These features of the system were internationally recognised when several papers describing PWB/UNIX were accepted for the Second International Software Engineering Conference in 1976. In May 77 the first (PWB/UNIX 1.0) manual was published and permission for export (outside Area 90) was granted in July. Promptly several PWB/UNIX licenses were issued to users outside the Bell System.

After the announcement of the MOPS committee report on the standardisation of UNIX,[18] it was decided that the PWB group would release UNIX/TS. UNIX/TS was intended to replace USG/UNIX while PWB/UNIX and would inherit bits of both (and others). A small update package (PWB/UNIX 1.1) was released in November 1977. Also in this time frame, some staff moved from PWB to USG to initiate the gradual centralisation of UNIX development.

Since UNIX/TS was to be based on the significantly different v7, and PWB 2.0 was to be based on TS, another update package was issued in May 1978 bringing PWB/UNIX to release 1.2. It

---

17. Indeed, the PWB group claims to have received the first copy of C-UNIX outside the Research Center.
18. See USG section.

contained the new C compiler, lint, the new shell and other software. The idea was to make the transition to 2.0 easier. PWB/UNIX staff claim that this was the best PDP-11 UNIX released, quite considerably better performing than the equivalents from Research and USG.

PWB/UNIX 2.0 was launched in June 1979. It was derived from UNIX/TS 1.1 and contained the new file system, the C compiler and some prototype graphics. After this, the Programmer's Workbench effort was absorbed into the UNIX Laboratory.

Although the most useful general features of PWB/UNIX were incorporated in Release 3.0 and its successors, some peculiarities of PWB/UNIX, particularly its support for Computer. Center type applications (servicing a very large user community with a variety of devices and needs) were not included (perhaps, because UNIX was being prepared for outside release). This void was filled by special Computer. Center releases of UNIX.

Although later releases from the UNIX Laboratory incorporated the most useful general features of PWB/UNIX, they failed to provide the 'utility' type service (for a very large user community with diverse needs) needed by the Computer. Centers at Bell Labs. This was because general UNIX was being readied for release outside the Bell System and the all the software needed for a Computer. Center type system could not be easily released outside, hence special systems had to be put together to fulfil the needs of the Computer. Centers.

3.3.4 Columbus Operations Systems Group

One of the first uses of UNIX outside the Research Center was in Switching Control Center Systems. The PDP-11 assembler version was used for maintenance data collection from electronic switches, being preferable to DEC's own system at the time. By the time the C version came out, the Operations System Group (OSG) at Columbus, Ohio already had a good working relationship with the Research Center staff.

Most of their enhancements to UNIX were in the real-time area and it wasn't long before other similar projects started adopting the SCCS version of UNIX and it soon became a standard in Columbus. By this time USG was maintaining a standard version of UNIX, but that was aimed primarily at computer center type applications and did not satisfy the needs of several projects, particularly in the real-time area, hence CB/UNIX continued to flourish.

In 1977, the MOPS committee recommended the use of UNIX/RT as *the* standard real-time UNIX. However this was not acceptable to the OSG at Columbus and a number of other projects because, in their opinion, the MERT based UNIX/RT would take a while to become as reliable and mature as USG UNIX (on which CB/UNIX was based), so development on CB/UNIX continued. Seeing this, Roger Faulkner, a supervisor at Bell Labs, Indian Hill, Illinois combined

the features of UNIX/TS and CB/UNIX to make UNIX/TS+ (also known as TS augmented) which became very popular.

Official work on CB-UNIX stopped when the desirable features of CB/UNIX were eventually offered by the mainstream UNIX with the release of System V.[19] However there are still some installations in Columbus which use modified versions of CB/UNIX.

In the early days CB/UNIX was mostly released informally, however, since some WECo products were built on top of it, manuals and other documentation had to be prepared. OSG followed the Research Center practise of naming the operating system version after the current manual. Editions 1.0, 2.0, 2.1, 2.2 and finally 2.3 were published in mid 1977, January 1979, January 1980, January 1981 and mid 1981 respectively.

All the CB/UNIX versions were for the PDP-11 (hence its demise for projects switching to the VAX) but made significant contributions to UNIX in the following areas.

- **Line Disciplines.** Different protocols were supported as were different terminal types. There was also a software multiplexer and character DMA.
- **Interprocess Communications.** Named pipes, messages and semaphores were implemented. Also signals and MAUS (multiply accessible user space) were developed.

In addition it made contributions in the areas of power failure recovery, process locking and logical file systems.

3.3.5 University of California at Berkeley

The academic community first became interested in UNIX as soon as it was first publically announced at the ACM Operating System Symposium in 1973. The Computer Science staff at Berkeley immediately wanted the system but did not have a machine to run UNIX. They soon convinced the Maths staff that purchasing a PDP-11, to share between the two departments, would be a worthwhile investment and by January 1974 such a machine was obtained.

The then current version of UNIX (Version 4) was locally[20] installed but the Maths department wanted to use DEC's RSTS so the two systems were rotated on the same machine. Despite this

---

19. Although this process of centralisation was official started with the release of 3.0, some bits of CB/UNIX did occasionally find their way into mainline UNIX before that.

20. Up until then, Thompson had been personally involved with each installation.

inconvenience, the UNIX system became very popular with the students and several projects started moving over to it. The **Ingres** project was one of the first to do so.

The added load proved too much for the machine so another PDP-11 was purchased and the latest version of UNIX (by then v5) was installed. **Ingres** moved to it straight away and was refined enough to be successfully distributed in autumn 1974.

Even though **Ingres** had moved, the first PDP-11 was still heavily loaded. To satisfy the need for more computing power a PDP-11/70 was obtained. This coincided with a one year sabbatical to Berkeley by Ken Thompson. He helped maintain the system while he was there. At his departure, two graduate students (Bill Joy and Chuck Haley) took over the role of kernel expert and slowly felt their way through the source code.

Earlier they had refined a Pascal system that Thompson has written during his sabbatical. The system proved very popular and was being requested by external sites. As a result, Joy put together the Pascal system and some other work and shipped out 30 free copies as "Berkeley Software Distribution".

Joy felt constrained by the line editors available on UNIX at the time, so he set about writing a full screen one as soon as Berkeley had suitable hardware available. This evolved into the popular **vi**. In mid 1978 another distribution was put together which included the editors and amendments to the Pascal system. Joy himself did most of the distribution work for 2 BSD.

Soon the PDP-11/70 was felt to be inadequate and the quest for a larger machine resulted in the purchase of a VAX-11/780 with DEC's own VMS operating system. Used to the pleasant environment of UNIX, the department did not want to change to VMS so they obtained the only available UNIX for the VAX: Bell's 32/V. This system did not take advantage of the hardware capabilities of the new machine so a student (O. Babaoglu) in the systems department set about implementing a virtual memory system on 32/V. With help from Joy the work was completed in January 1979. By December, all the 2 BSD software had been ported to the VAX and the complete system - the first complete BSD/UNIX system - was shipped as 3 BSD.

When ARPA decided to standardise the operating systems used at its network nodes, it chose UNIX due its proven portability and general popularity. Berkeley was quick to seize the opportunity and offered to develop an enhanced 3 BSD for ARPA use. By April 1980, ARPA had formally agreed and, in response to this, the Computer Systems Research Group (CSRG - where the UNIX work was done) formalised its UNIX effort and legal arrangements were made with AT&T to more freely distribute BSD/UNIX.

4 BSD was released in October 1980 and incorporated new **job control, auto reboot, 1k file system,** a new **Pascal** compiler along with **Franz Lisp** and enhanced **mail.** 150 licenses of this system were granted for 500 machines.

The popularity of 4 BSD/UNIX meant that it was heavily used and some performance problems surfaced. To answer these a tuned up version, 4.1 BSD, was launched in June 1980. More than 400 copies were distributed in its short lifespan.

The successful launch of 4.1 BSD attracted more ARPA funding, much greater than before. To better organise this, formal goals were set by ARPA and an inter-company steering committee was set to direct the development. Among the requirements were to support the ARPA communications protocol TCP/IP and a faster file system.

By April 1982, the first implementation of the interprocess communication mechanism was completed and was locally distributed, for feedback, as 4.1a. Concurrently Kirk McKusick continued working on a new fast file system. This work was completed and integrated with the 4.1a changes by June 1982. But because of the large overhead involved in transferring from 4.1a to 4.1b (as it was known), 4.1b was not distributed, even locally.

As work for 4.2 dropped behind the ARPA schedule, due to organisational changes necessitated by the imminent departure of Bill Joy, a provisional release, 4.1c, was shipped out in April 1983, to a few sites to tie them over until 4.2. Eventually, in August 1983, 4.2 came out incorporating a cleaned up version of 4.1c with networking support and other work. At this the successor of Bill Joy, Sam Leffler, also left CSRG and Mike Karels was invited to lead the BSD/UNIX effort.

In December 1984, Kirk McKusick joined Karels to continue the development of BSD/UNIX and to solve some of the performance problems introduced by 4.2. Originally aimed for summer 1985, 4.3 came out in summer 1986.

Work on BSD/UNIX is continuing at a furious pace though ARPA's support of CMU's MACH operating system would suggest that its support of BSD/UNIX is likely to reduce in the near future.

3.4 LICENSING

Amongst the factors put forward for the success of UNIX are (1) the early lack of marketing pressure and (2) cheap dollar cost [MCI86]. However, it appears that there *has* been a marketing strategy right from the start, though initially it had to play second fiddle to the wishes of the research centre staff.

By the early 1970's AT&T was already shipping out large quantities of software. Most of this was sent out as is, with only a covering letter (not even a licensing agreement). This resulted in an uncontrolled flow of software, once it got out of the door, for example snobol.

Realizing that software had marketing potential, Dick Shahpazian[21] set about evaluating 10-12 packages, some of which had already gone out. He considered patent as well as technology licensing. He eventually formulated a licence agreement, but the terms and conditions were so complex that a few educational institutions actually turned down licences.

Concurrently, a survey was conducted to determine the marketing potential for UNIX. Aiding him in the survey were some members of the Research Center, they suggested that he would be hard pressed to to get anyone to pay more than $2,000 for it. Shahpazian's commercial survey, however, indicated that some companies would be willing to pay up to $25,000 for it. Eventually it turned out that *none* of the companies that said UNIX was worth $25,000 bought a licence indicating that the industrial market wasn't quite ready then to accept un-supported UNIX.

The licensing agreements with educational institutions and other non-profit organisations were very different to the commercial ones. Initially, the Research Center was so keen to get it out of the door, that they were against charging even a service fee but as the volume grew, they gave in. The first educational licence was granted, in October 1973, to Columbia University and Ken Thompson personally installed the system. Educational licences for the latest Research versions are still offered for nominal charges but Thompson no longer goes out to install them!

The Children's Museum in Boston was the first non educational recipient of UNIX in October 1973 and The Hebrew University of Jerusalem was the first organization outside the US to obtain a licence, in February 1974. Queen Mary College in London was granted a licence in May 1974 and the Rand Corporation became the first commercial licensee, in July 1974.

Since UNIX was offered as is,[22] there was a big market for supporting UNIX (specially with full source available on-line) and a number of software houses jumped at the opportunity, Interactive Systems[23] being the first one. Their success and the growing popularity of UNIX forced the big computer manufacturers to take notice and now *all* the major ones are UNIX licensees.

---

21. At the time Assistant Manager, Patent Licensing.

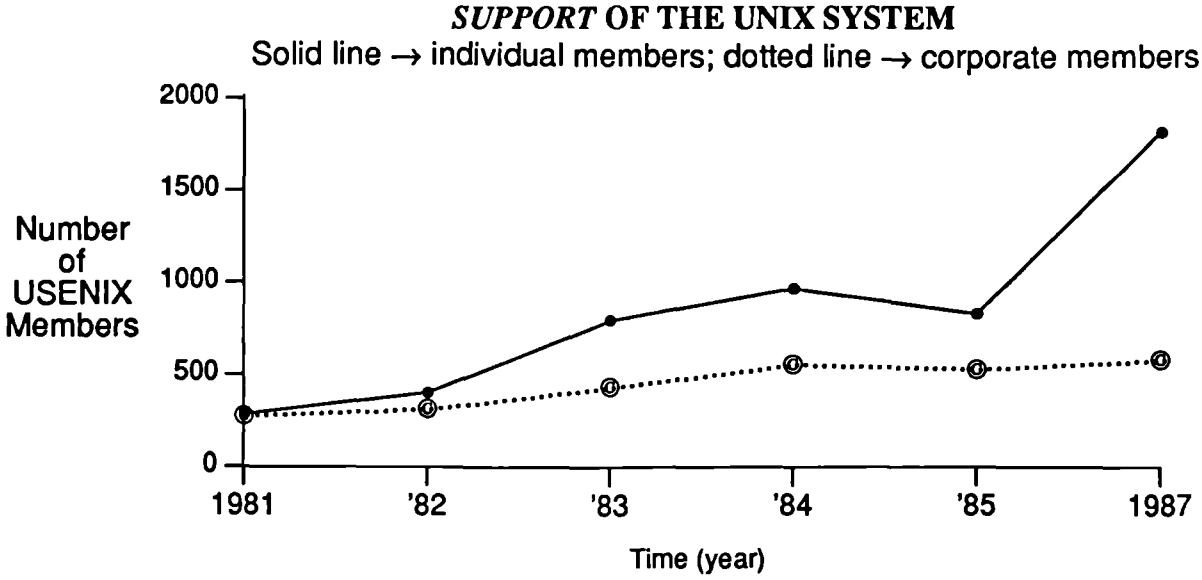
22. Until System V, in 1983, whence full support was given.

23. Under Peter Wiener. Intriguingly, Peter Wiener was also at Rand when they got their licence, so he was the first commercial user and the first systems house user.

Before software houses jumped on the UNIX bandwagon, users had to support themselves. Also the Bell Labs policy of offering free educational licenses meant that UNIX spread like wild fire in the universities. The UNIX philosophy of on-line source code and information sharing spilt onto the user community and soon they were sharing ideas and code alike and local user groups started springing up. Seeing that everyone would benefit from a global group Lou Katz (at Columbia) and Mel Ferentz (at the Brooklyn College of the City University of New York) set up */usr/group*, an organization dedicated to the sharing of information between licensed users of UNIX. They published a bi-monthly newsletter *UNIX NEWS*, the first issue of which came out in July 1975.

In July 1977 the name of the newsletter was changed to *;login:* and in June 1979 the *USENIX Association* was formed (by the above two people!). With the increasing popularity of UNIX, lots of manufacturers had joined /usr/group and *users* felt the need for a separate body to freely exchange ideas. USENIX was meant to fill that void.

40 people attended that first UNIX user meeting in New York (in 1974), now more than 2,000 attend. In 1981 there were 287 individual and 265 educational/corporate members and by 1986 there were 1815 and 569. The growth in membership is illustrated below:



Chapter 4
PROJECT METHODOLOGIES

"Intelligence ... is the faculty of making artificial objects,
especially tools to make tools."
(BERGSON)

4.1 INTRODUCTION

This chapter describes the tools and techniques used in this project to arrive at the models presented in the next chapter.

The structure of this chapter

After reviewing the different metric measures available for the desired process and product attributes and discussing their data requirements, this chapter describes the configuration management procedures used by the various UNIX groups. It then uses a discussion of the availability and survival of the required data sources and other resources to present a justification for the choice of metrics selected to model the UNIX evolution process. Finally, the computational tools and techniques used to extract the data from the data sources and construct the models are described.

4.2 SURVEY OF METRIC MEASURES

The fundamental 'law' of Program Evolution (see Chapter 2) states that software evolution "is subject to a dynamics which makes the programming process, and hence measures of global project and system attributes, self-regulating with statistically determinable trends and variances". This project is attempting to discover if the UNIX evolution process has been subject to the same dynamics.

In previous studies of this kind [LEH69], [HOO75], [CLA78], [CHO81] and [KIT82], some or all of the following attributes have been observed to display solid statistics:

- size
- complexity
- work-rate
- release content

This section discusses the various ways of measuring these and other attributes and their data requirements. Each attribute is discussed in turn.

4.2.1 Size

Traditionally size has been the dominant attribute of a program; it is easily calculated after completion, all effort models are in some way related to program size and programmer productivity is usually measured in terms of size per unit effort. Over the years a number of size metrics have been suggested and since this study was primarily concerned with the human/program interface (as opposed to the program/machine interface) this section will concentrate on program source rather than object code.

Lines of Code

In the olden days, to approximate the bulk of a program, one simply counted the number of punch cards containing the source. This measure survives today as the "lines of code" (LOC) metric. There are several ways of counting the lines:

1. The simplest is to count *all* the source lines as if the card reader were reading the punched cards.

2. However, blank lines or comments do not affect the function of the program since they are internal documentation and are not as difficult to construct as the 'real' source [CON86]. Hence the second way of measuring the size of a program is to count only the non-comment and non-blank part of the program. For free-format languages, this would include all lines that were not totally blank or comment (i.e. all lines that contained any 'real' source would be counted). This is the definition of size that is most commonly used by researchers, for example [BOE81].

3. A further refinement is to only count lines which have executable statements in them, since they are the ones that perform the *function* of the program, the rest are simply data declarations. This is not as popular as the second measure since writing data declarations does require a non trivial amount of effort.

All the above metrics require access to the source code, preferably in machine readable form. While a simple counter will suffice for the first. The second metric requires some intelligence to determine if a line is purely a comment or blank. The third requires further intelligence to separate the executable statements.

Token Count

Modern, free-format, high level languages like Pascal, ALGOL and C allow the programmer to write several executable statements on the same physical line, for example a programmer may write:

```
write a; write b;
```

while another one may write:

```
write a;
write b;
```

both are identical in execution, yet one section is twice the size of the other in a 'lines of code' count. There are several ways of getting around this problem:

1. The first is simply to count the number of *characters* in the program source, ignoring the lines. This automatically results in a higher number for lines with a heavy content but introduces a problem of its own: large identifier names. Writing large symbol names does not require more effort than small names, yet increases this count significantly, hence this metric is not very popular with researchers.

2. The second is to count the number of program *statements*. This metric is not affected by large identifier names but it has the drawback that statements can vary greatly in complexity themselves. It has been used in the past by some researchers, particularly for systems which are not stored in 'modules' or other physical units but is not so popular now.

3. Counting tokens solves the problems described above. Instead of counting lines or statements, we count compiler 'tokens', i.e. items that the compiler regards as atomic (e.g. identifiers, operators and reserved words). This automatically results in a complicated line being "bigger" than a simpler one. The most well known such measure was devised by Halstead of Purdue University [HAL72], [HAL77]. His metrics for *size* and *volume* of a program are derived from counts of the total occurrences (and number of unique) of operators and operands. These metrics have been the subject of much research [BEL74], [FUN76], [ELS78] and [CUR79a]. Most of these conclude that Halstead's metrics do not give any more information than LOC or statement counts. Recently, however, Prather

[PRA88] has suggested that Halstead's expected size metric may be a useful lower bound predictor.

Program source in machine readable form is needed in all three cases. A simple language independent utility is all that is needed to count the number of characters. A more intelligent source analyser is needed (for instance in 'C' something to count the number of ';'s would suffice) to count the number of statements. Even more sophistication is need for Halstead's metrics since the input has to be completely parsed.

Function and Module Count

Large systems are usually decomposed into more manageable subsystems. In turn these are further partitioned and so on until we get to the smallest unit which can be independently compiled. These are termed *modules*. The size of such large systems is usually given in modules rather than LOC. This results is smaller numbers which are easier to handle. The obvious problem is that the size of the constituent modules in a program may vary considerably [SMI80] unless there are strict guidelines on how to partition the system. Hence the 'ideal' size of a module has been the subject of some research [BAK72] [BEL74] (and so has scaling of a *project* [BAN88]). In spite of this, 'number of modules' is the favourite size metric in [BAS79], [LEH80] and other modelling studies.

A program is sometimes viewed as a model of some part of the 'real' world or as automating some manual function. Programmers find it easier to partition this large function into smaller functions than into modules or other physical units. A *function* is a collection of statements that perform a certain task and which logically go together. Research has shown that functions tend not to exceed a certain size there appears to be a limit to the number of things a programmer can concentrate on at one time [WOO81]. Also, it appears that programmers agree more on the number of functions required for a particular problem than the number of modules [BAS79].

A variation on this theme is a count of *function points* suggested in [ALB79]. With an aim towards finding a metric for estimating the effort required to produce programs he suggested counting the number of inputs, outputs, inquiries and files (termed function units) and then using a weighted sum. While he has demonstrated the usefulness of his metric in commercial applications, it is not popular in other departments because of the difficulty in (subjectively) assessing the function units.

The 'number of modules' metric for size is perhaps the easiest of the ones described here to measure since a file-list is all one requires (since the numbers are smaller, sometimes even non-machine-readable ones are !). Counting the number of functions requires source code and a tool intelligent enough to recognize function definitions. Alternatively, design documents may be

used if they have enough detail and are stored on-line under some change control mechanism.

4.2.2 Complexity

There are two aspects to software complexity: computational and psychological. The first is concerned with the program's interaction with the computer (such as algorithm efficiency and machine resources) and the second with the program's interaction with the programmer. Since the primary motivation was a desire to test the second 'law' of program evolution (see Chapter 2), this section will concentrate on the second aspect. Precisely defining software complexity had been a major challenge until [CUR79] proposed:

> Complexity is a characteristic of the software interface which influences the resources another system will expend or commit while interacting with the software.

and this has been widely accepted since. Over the years many metrics have been proposed to measure software complexity [BEL79] but only those which attempt to convey software's 'resistance to change' will be discussed. They may be broadly classified as follows.

Control Structures

The metrics which measure software complexity by assessing the control flow of the software can be further divided.

Decision count — The most well known of these is McCabe's cyclomatic complexity metric *V(G)* [MCC76]. Based on graph theory, the metric counts the number of distinct control paths in a program. His argument is that the higher the decision count, the more difficult it is to test. A number of people have suggested variations on this theme [CHE78], [BAS79a] and [MYE77] while [WIL72] had come up with a similar scheme earlier. Numerous criticisms of the theory, e.g. [SHE88] state that the cyclomatic complexity measure is not better than LOC as a predictor of complexity.

Reachability — Using similar rationales as McCabe, [SCH83] proposed metrics for the minimum number of paths in a program and the reachability of any node. The calculation of these metrics is awkward for large programs so [SHO83] proposed a technique for estimating them.

Nesting levels — [DUN79] shows that excessive nesting causes difficulty for a programmer to comprehend what conditions must hold true for a particular statement to be reached. [ZOL81] and [DUN80] show that

the depth of nesting and the average nesting level are useful complexity metrics.

Unstructured transfers — In his classic paper [DIJ68], Dijkstra stated that the quality of a program was inversely proportional to the number of 'GOTO's in it. He showed that direct transfers disrupted the programmer's perspective. Subsequently, Woodward et. al. [WOO79] proposed a metric, *knots*, to measure control flows which cross each other.

All of these metrics require machine readable access to program source and sophisticated tools to extract the desired information.

Composite Metrics

The idea behind these metrics is that it is very difficult to assess the complexity of a program by a single (simple) metric, i.e. the ones described above. The obvious approach is to try a linear combination of the simple metrics. Again, the most well known of such metrics is Halstead's *effort (E)* metric, defined in terms of unique operators (n_1), unique operands (n_2), total occurrences of operators (N_1) and total occurrences of operands (N_2) as:

$$E=\frac{n_1 N_1 (N_1+N_2)\log_2(n_1+n_2)}{(2n_2)}$$

He suggested that E gave the number of elementary mental discriminations to understand the program. He understood from [STR76] that the human mind had a limit to the number of mental discrimination it could make per unit time. This psychology assumption has been more thoroughly analysed in [COU83]. Halstead's formulae have been criticized by [KIT81], [SHE83], [SHO83] and others who show them to be no better than LOC for measuring complexity.

[MYE77], [HAN78], [BAK80] and [OVI80] have suggested other composite metrics based on McCabe's $v(G)$, Halstead's E and others.

Since composite metrics involve, at a minimum, calculation of simple metrics, they require at least as much data and resource as the simple metrics described above.

Interconnectivity

Since [PAR72], modularisation of software has been on the increase. Keeping pace with that, metrics have been suggested to assess the complexity of the interconnections between parts of a system. The idea behind that being: complexity is related to the proportion of the rest of the system the programmer has to understand to work on this part. This is true for both physical units

(e.g. modules) and logical units (e.g. functions).

Myers [MYE78] has suggested modeling complexity as two aspects: the strength of the module and the coupling between modules. He advocates having as much independence between modules as possible, suggesting that the complexity of the interface between modules is a good predictor of system complexity but has not, as yet, offered an operational definition.

Benyon-Tinker [BEN79] states that the effort needed to understand a program is related to the depth and breadth of the procedure calling tree. He suggests the following complexity metric:

$$C_\alpha = \frac{\sum_{r=1}^{r=m} n_r r^\alpha}{\sum_{r=1}^{r=m} n_r}$$

where n_r is the number of distinct nodes (function call hierarchies) at level r, m is the maximum depth of a tree, and α is a power-law index. For the system that he studied, he suggested a value of between 2 and 3 for α.

Others, e.g. [CHO81], have suggested counting the number of times a particular function is called by others and the number of other functions called by this function and then adding them up to get an indication of the total program complexity.

The central theme in assessing interconnectivity is counting the number of links a module (or part) has with other modules (or parts), whether through global variables or through function calls. [BEL75] have attempted predict whether (1) a change hits a given module or (2) another module is affected by the change by introducing distributions. Their work was a development of [EMD71].

All these metrics require machine readable access to source and sophisticated analysers.

4.2.3 Process Attributes

All the metrics described above are *product* metrics, that is they have described and are calculable from the software. They do not take into account the history of the software or how it was produced. The metrics presented below describe *process* attributes and are concerned with how the software was produced. Very little research has been targeted at measuring the quality and complexity of the process. The notable exception being the Evolution Dynamics studies initiated by Lehman and Belady in 1969. All the following metrics were suggested in one or the other of these studies.

Work-rate

Effort expended per unit time or *work input* has been measured by counting the number of modules handled (i.e. worked on) during the specified time period (i.e. if module 'a' was changed during the interval, it is counted). It has also been measured by counting the number of changes made to the program and by counting the different module versions handled (i.e. if module 'a' went through three versions in the interval then the count is incremented by three).

The number of changes made to the program can only be counted if some records are kept of all source changes. Versions can only be counted if the source is under a version control system. Similarly, the module handlings metric requires the source to be under a source control system.

Release Content

As yet, there is no precise way of measuring release *content* [LEH80], as any definition must take into account the size, complexity and the inter-relations between the system and the code changes. However, Lehman has used net incremental growth (between releases) as an indicator of release content. Kitchenham [KIT82] used the ratio of different module versions handled to the modules handled as aid to determine the difficulty faced by the programmers to arrive at the desired content.

To calculate the incremental growth, we need only have the size at the two extremes of the required interval. To use Kitchenham's metric requires the source to be under version control and the records to be machine readable.

4.2.4 Other Attributes

Previous evolution dynamics studies have modelled the evolution of systems in terms of the attributes described above (though not necessarily with the same metrics). Apart from error-rate, they have not, however, systematically examined the attributes discussed below. It would be interesting to see how these attributes have changed during the lifetime of UNIX since the fourth 'law' of program evolution claims that the work-rate of a programming process remains invariant throughout the life-time of a project, *irrespective of staffing and technology changes*.

Staffing

In various cost estimation models (see Section 2.2) the following measures of staffing have been suggested as cost drivers:

- Number of technical staff

- Number of years team has been programming
- Number of years experience in particular programming language
- Number of years programmer has been in the team and company
- Number of years experience of similar software construction and hardware

Measuring these require access to detailed organization charts and career histories. Timesheets would also be needed if effort (in person-months) is to be calculated.

Programming technologies

The effect of programming technology on programmer productivity has been the subject of much research [PUT78] and [PUT84], [BOE81], [BOE84], [DRU82]. However, there are not many metrics for *measuring* programming technology. The following have been informally suggested in [CON86]:

- Use of top-down development techniques
- Use of structured programming
- Use of design languages and systems
- Use of version control systems

assigning the values of 1 and 0 to the metric depending on if they are or are not used. Alternatively, a numeric value could be assigned, representing degree of usage.

These require access to operations manuals, programming standards documents, integration guides etc. for the target organization.

Error-rate

Even software produced by the best programmers using the latest programming technology contains errors. Software is considered to have errors if it does not do what is required of it. *Predicting* the number of errors has been the subject of much research [AKI71], [MOT77], [HAL77], [OTT79], [POT82], [MUS75], [MUS80] inter alia. This investigation was, however, concerned with *measuring* the defects, so the prediction metrics here will not be discussed here. Conte et. al. [CON86] proposed a number of ways to assess the defects in software, throughout its life-cycle:

- Number of changes required in design. This is a subjective measurement requiring the analyst to judge the number of 'separate' items changed during the design phase.

- Number of errors discovered. This is a count of all errors discovered during coding and testing and use. Another useful metric would be to count program crashes to measure the reliability of the program.

- Number of program changes. The rationale behind this metric is that a program change is the result of an error discovered. It is defined as a change of a contiguous set of statements that represent a single abstract action [DUN80]. A refinement of this would be to count the number of lines changed as this would also reflect the magnitude of the change.

The above metrics require access to the equivalent of design amendment forms, error reports, code change forms and histories of source control and version control systems.

Functionality

Functionality is a very nebulous concept and no well known metrics exist to measure it, however feature lists would seem to provide an indication of functionality. Therefore possible metrics could be:

- A count of the number of manual entries.

- A count of the thickness of the manual and other documentation (as this is sometimes used as a complexity indication).

These measurements must, however, be used with care, since they do not adequately measure desirable features such as simplicity and generality.

Access to machine readable copies of manuals and other documentation is needed.

4.3 SOURCE MATERIAL

4.3.1 Ideal sources

The previous section discussed various ways through which the desired process and system attributes could be measured. The table below summarizes the source material required by those metrics.

DATA SOURCE	FOR ATTRIBUTE
source listings	most size and complexity metrics, growth-rate
design documents	may suffice for some size and complexity metrics (e.g. function counts
manuals	functionality, release dates, may also help in staffing
release documents	release dates, some size metrics (if they have file-lists), work-rate (in case lists of changed files are provided)
bug report system records	error-rate
source control system records	work-rate, release content, error-rate
time-sheets	staffing, effort (in man-months)
career histories	staffing (experience of staff etc.)
organization charts	staffing
operations documents	programming technology (use of structured programming etc.)
user, licensee lists	usage (number of installations, distribution of user organisations)

4.3.2 Configuration Management in UNIX Groups

This section describes the configuration management procedures used in the UNIX groups under study and hence speculates on which data sources, from the ideal ones described above, this study could have hoped to obtain. Each UNIX centre is discussed in turn.

Research

The technical staff at the CSRC have always had an informal approach to configuration management when programming. Since their charter is to conduct fundamental and applied research in different aspects of computer science, they feel that they need creative freedom which most structured configuration management procedures would only inhibit.

Hence, there are no forms to fill when changing code, though it is understood that some programmers kept simple records of changes made to the part of the system they were responsible for. Most of the programming at the CSRC is done by the researchers for themselves, so there is little incentive to maintain different versions of code, hence version control systems are not in use. Similarly, there is no contractual obligation to record the wishes of external users so there are no 'problem report' systems. Indeed the researchers are not even asked to account for partitioning their time, as all work in the CSRC is allocated a single project number.

The rate of evolution of the system was controlled by the programmers themselves (see Chapter 3) and a new 'release' was identified by a new edition of the programmers manual. The system was initially distributed by dumping the contents of the CSRC machine onto tape and sending the tape out with a covering letter and minimal documentation (installation guide and manual) but a list of installation was kept. New editions of the manual were prepared when it was apparent to staff at CSRC that the manual no longer reflected the state of the CSRC system, this was usually done informally and no official release documents as such, were prepared. This strategy still exists today but the distribution was handed over to the Computing Library at Bell Labs. which set up slightly more formal arrangements based on distribution tapes supplied by the CSRC, for the releases v6 - v8.

It is clear the CSRC has opted for a compromise 'system' which allows rapid code prototyping rather than providing sound configuration management.

Therefore the best that this study could have hoped for was to get hold of:

- all editions of the programmers manual
- all prepared distribution tapes (v6-v8)
- systems representing manual editions 1-5 and 9.
- private records of changes
- private lists of installations

and accepted that a continuous picture of Research UNIX was not be re-constructable, snapshots at release points would have to suffice.

USG, UDL and ATT-IS

One of the primary reasons for creating the UNIX Support Group was to filter requests from users to the Research Center. Hence mechanisms were set up to record these requests. Initially a form was supplied at the back of the *UNIX Newsletter* (published by USG) to allow users to submit (i.e. send by internal mail) **trouble reports** (TRs) to USG. Each month, summaries of these TRs were published in the Newsletter. These summaries included a section on what action USG was going to take. Later on, this was automated, allowing remote users to login on to the USG machine and enter in the TR themselves. When the PWB effort was absorbed into USG (becoming UDL), a planning department was formed which replaced the TR system with the more formal MRCS (originally developed at PWB, see below). This later on evolved into the Change Management Tracking System (CMTS), providing for the increased number of states in the MR life-cycle.

The other main reason for the formation of USG was to steady the environment; the CSRC system was changing too rapidly for other groups to keep up. To keep 'old' versions of the source available until users had caught up, USG master source was kept under version control as soon as SCCS (from PWB, see below) was released. This also aided traceability and provided a record of changes made to the source. As UNIX development grew, the source control was brought under the umbrella of the CMTS, described above, to provide an integrated change management system.

USG/UNIX systems were released more formally, to its customers, than Research UNIX. Hence plans were published, new release announcements were sent out to existing customers and interested parties and detailed release descriptions were prepared to go with the distribution tapes.[1] These mechanisms became more formal when planning and support departments were introduced in the UDL.

Work was specifically allocated to members of USG, and these allocations were occasionally published in the newsletter. Internal (to the Bell System) distribution was handled by USG itself,

1. Some applications were available on-line to customers who could ring up the USG machine and copy (or mail themselves) particular code from the "treasure chest".

at least initially, so installation lists were maintained. In an effort to teach UNIX to non-experts, courses were organised and documentation prepared. When UNIX was shipped outside the Bell System under a Western Electric licence, formal documentation had to be prepared. Hence a system description was written [BRA76], but was never updated to keep pace with changes to the system. Since the commercialisation of UNIX, the "support" for UNIX has been provided by different parts of AT&T, for instance, licensing by AT&T Technologies.

Therefore, following official records were kept, at some point or another, by USG, UDL and ATT-IS, which this study could have hoped to access:

- Distribution tapes and corresponding manuals for *all* releases
- Release descriptions, announcements. Planned schedules
- Work breakdown allocations
- On-line trouble-report and modification report databases
- Complete on-line SCCS source database
- User and installations lists

PWB

Programmer's Workbench UNIX arose out of a need to preserve a stable *programming* environment in the light of frequent target hardware and target software changes. Since it was meant for very large development teams (hundreds), it is not surprising that the PWB groups had the most ordered configuration management from the original UNIX groups. Since they had developed tools like SCCS and MRCS to control the evolution of their target software (i.e. the software they were using PWB to produce), it was not difficult to use these to control the evolution of PWB/UNIX itself.

Well versed with supporting the development of large software systems, the PWB group were organised in announcing new releases and preparing proper release description documents for their versions of UNIX. This was also necessary to minimise disruption to general users, since PWB/UNIX also provided a computing service to other users in Piscataway.

Hence it was reasonable to expect to get hold of at least as many possible data sources as for USG, listed above.

CB-OSG

Although the Operations Systems Group at Columbus was a pseudo research group, it was supporting the Switching Control Center System and was being shipped out as a Western Electric product so documentation had to be prepared for it. Since it also provided a computing service to users in Columbus, releases were controlled and release descriptions were prepared. However, like the CSRC, little effort was put into recording change requests and keeping the source under version control.

Therefore, it should have been possible to get hold of:

- Distribution tapes of all released versions of CB/UNIX and their manuals
- Release descriptions

4.3.3 Data Collection: The Project Database

This section describes the successes and failures in locating and obtaining the data sources identified in the previous section.

Research

UNIX gurus are commonly portrayed, for example in USENIX or EUUG conferences, as unruly, casual and disorganised. This is somewhat borne out by their record keeping record. The researchers were so concerned with pushing at the frontiers of operating system technology (in writing UNIX) that they did not bother to keep the 'old stuff'.

Hence in the CSRC archives,[2] only copies of the sixth and seventh edition distribution tapes. A complete copy of 'standard' eighth edition tape(s) was kept on-line on one CSRC machine which was made accessible to this project. There isn't an official ninth edition distribution tape as such, however a copy of their master source at the time of the preparation of the manual was made available to the study. Since the Research manuals describe the state of the system as-is (on the date of publication), a copy of their master sources should be a reasonable approximation. Getting hold of UNIX systems representing each of the five versions before v6 has turned out to be a major challenge since no dumps from that era survived in the CSRC archives. The author was

2. In reality, their 'archive' is simply a room adjoining the UNIX room where the dump tapes and other precious material are kept. There is virtually no security, indeed, there isn't even a list of what is stored in the room!

Please see addendum for more of section 4.3.2.

able to locate only one UNIX system corresponding to the v5 era,[3] and none before that. Since Research UNIX did get outside the CSRC, from v3 onwards, there was a chance that someone might have preserved it but there was no positive response to several requests in electronic newsgroups, conferences etc. However, some old paper tapes and DECtapes of unknown content were discovered underneath the floor-boards in the UNIX room.

With a track record of poor record keeping, it came as a surprise when the first six editions of the manual were found neatly bound in the UNIX room. The seventh edition is generally available and the author has obtained copies of eighth and ninth editions as well, making a complete set.

Getting hold of other information on Research UNIX has been less fruitful. Private records of changes made to the system were kept (e.g. **ken** kept records of changes made to the kernel when UNIX was identifiably 'his baby', in the early days) but got truncated every time the file got too big (supposedly fairly often) and deleted at machine changes. So, changes information is irrecoverably lost. Similarly system logs containing records of system crashes etc. have also been lost over the years. However, a list of the first 25 licensees was obtained from Thompson (who was handling the distribution himself at that stage). Subsequently licensing was handled by a Western Electric division and then AT&T Technologies (AT&T-T). AT&T-T have detailed records but were not able to send them to the investigator in time for inclusion in this study.

USG, UDL and AT&T-IS

Since the charter of USG was to support UNIX, it would be reasonable to expect them to maintain sound archives. In fact, as described earlier, they did keep records, however the records have not survived very well due to the several organisational changes, hardware changes and location changes.

Amongst the most important information lost in the various moves are the on-line SCCS records of source changes. The current system manager at the AT&T-IS facility at Summit explained that it was very expensive to keep old records, for instance just to take *one* dump of the system there requires 500 tapes! There are some records of changes made to the system since "sometime into System V" but they were not made available to this study. The MR database, surprisingly, has

3. This system was used to drive the displays at the entrance of the Bell Labs complex at Murray Hill. Since the system was used by only one person for a single static purpose, the system did not need upgrading.

survived; however the investigators copy was lost (along with all the interview notes and other valuable information) during an office move. Hence it was not possible to reconstruct an authentic continuous picture of the evolution of supported and commercial UNIX.

Since a central repository did not exist, the required distribution tapes had to be sought from various individuals, hence a complete sequence was not possible but the ones for Generic 3.0, UNIX/TS1.0, Releases 3.0, 3.0.1, 4.0, 4.2, 4.2.1, 5.0, 5.0.1, 5.0.3, 5.0.5, System III and System V Releases 1.0, 1.1, 1.2, 2.0, 2.2 and 3.0 were obtained. Ex-USG staff (especially management) were more successful at keeping paper documents like manuals and release descriptions and the following were kindly lent to this investigation: release descriptions for Releases 3.0, 4.0, 4.2, 4.2.1, 5.0, 5.0.1, 5.0.3, 5.0.5 and System V Releases 1.0, 1.1, 1.2, 2.0, and 2.2 in addition to manuals for Generics 2.0, 3.0 and UNIX/TS 1.1. Some published schedules were also obtained but neither user or installation lists nor the work allocation schedules were traceable.

An almost complete collection of the *UNIX Newsletter* was obtained which helped to fill in the many gaps left by the lack of information above.

PWB

The PWB staff were eventually incorporated into the UDL, so their on-line records were merged with UDL's and were also lost (along with USG/UDL ones). However, PWB/UNIX 1.0 was licensed outside the Bell System and 2.0 was also sent out. So this investigation managed to get hold of distribution tapes for them. In addition, release descriptions for 1.0, 1.2, 2.0 and 2.1 and manuals for 1.0, 1.1, 1.2 and 2.0 were discovered with ex-PWB staff.

CB-OSG

The master source of CB/UNIX was kept under some source control (their own version of SCCS, not the standard one) but the records were lost when the CB-OSG group was disbanded and staff dispersed. Fortunately, some staff were traced and the following documents obtained: manual for 2.3 and release description for 2.3. In addition private records revealed manuals for 1.0 and other release descriptions but they could not be obtained. Distribution tapes of releases 1.0, 2.0, 2.1 and 2.3 were also obtained.

4.4 DATA EXTRACTION

The last section listed the distribution tapes obtained by this study. This section describes the problems encountered in trying to extract information from these tapes.

Several steps are required to successfully 'read' from tape (or other device such as disk-drive). First, access to the right physical device has to be obtained. Then a device driver, to interface

Please see addendum for more of section 4.3.3.

between the device and the system, has to be found. Then (assuming the bits can be loaded from the tape) the type of the program used to write the tape has to be determined. Finally, a compatible program to read the tape and reconstruct the original directory hierarchy (containing the source tree) has to be found. Each of these will be discussed in turn.

4.4.1 Devices

Since the official archives were very poor, this investigation had to make do with whatever it could get its hands on. The UNIX related software that this study obtained was stored, in probable chronological order, on paper tape, TU59 DECtape and RK05 disk units in additional to 800 bpi and 1600 bpi magnetic tape.

In a state of the art setup like at CSRC, when new technology is introduced, frequently the old is thrown away, this is especially the case when the new is not only better but also cheaper. Hence no paper tape or DECtape readers could be found at the Labs. After an extensive search[4] an almost functioning DECtape unit was discovered in Canada. The local gurus succeeded in reading the tapes but did not find anything of relevance to this study.

RK05 units, although abandoned by modern installations, are still used by 'one-off' set-ups created earlier. One such installation was discovered in the Labs., not far from CSRC, greatly facilitating the reading of the RK05 disc packs. Fortunately, one of the packs turned out to a v5 era research system and the other a USG generic 3.0 system.

Since magnetic tape is still the most popular off-line storage medium, locating a 800 bpi tape unit was much easier, even though the vast majority have now switched to 1600 bpi.

4.4.2 Loading

After locating the required device, the next step is to load the information from the medium (tape or disk) onto the computer. On UNIX systems it is usually straight forward to transfer the bits stored on the medium to on-line storage, using a utility such as **dd**.

Difficulties are encountered when the data on the tape (or disk etc.) is corrupt. This was the case with one of the RK05 disk packs, but one of the systems support people helped, by writing a short

4. Locating relatively obsolete technology is difficult because most of these devices are installed in stand alone systems whose users are often not computer scientists and not interested in updating it. These people are not likely to come in contact with the computer science community and getting across to them is next to impossible. This is further complicated by the fact that sometimes the staff do not even know what they have!

program to bypass the corrupted blocks but copy the rest onto tape. Since only one block turned out to be spoilt, most of the system could still be reconstructed.

Unfortunately, some other potentially valuable tapes could not be read due to severe corruption. Hence sources for CB/UNIX 1.0 and some early USG releases (of unknown vintage) could not be analysed.

4.4.3 Reading

Once the tape is loaded, the right archiving program, to reconstruct the directory hierarchy from the bits, still has to be found. The most popular tape manipulation programs used in the post VAX UNIX world are **tar** and **cpio**. Some of the information written by these programs is binary, so not portable between different machines but options can be used to write everything in *ascii* to achieve portability, and this was done for all non-VAX tapes that were obtained. So, there was no problem with reading most of the tapes that were written after about 1980. Trial and error was needed to find out what program was used to write the tape. This includes all the BSD tapes, all (bar the ones described above) supported UNIX tapes and CB/UNIX 2.3. The Research versions v8 and v9 were already on-line.

The v7 tape was originally prepared for the PDP-11 but we were able to reconstruct the directory hierarchy by using a modified version of the v7 **mount** program that was maintained privately by one of the UNIX staff (Ritchie). The older v5, v6 and Generic3.0 tapes were simply dumps of the systems from PDP-11 days so there was no obvious way of reading them. After much searching, one of the systems staff came up a program that could read the old dumps and list the files in a given directory or print a given file. A simple script was quickly written to drive this program to traverse the whole file tree stored in the dump and reconstruct the corresponding on-line directory hierarchy.

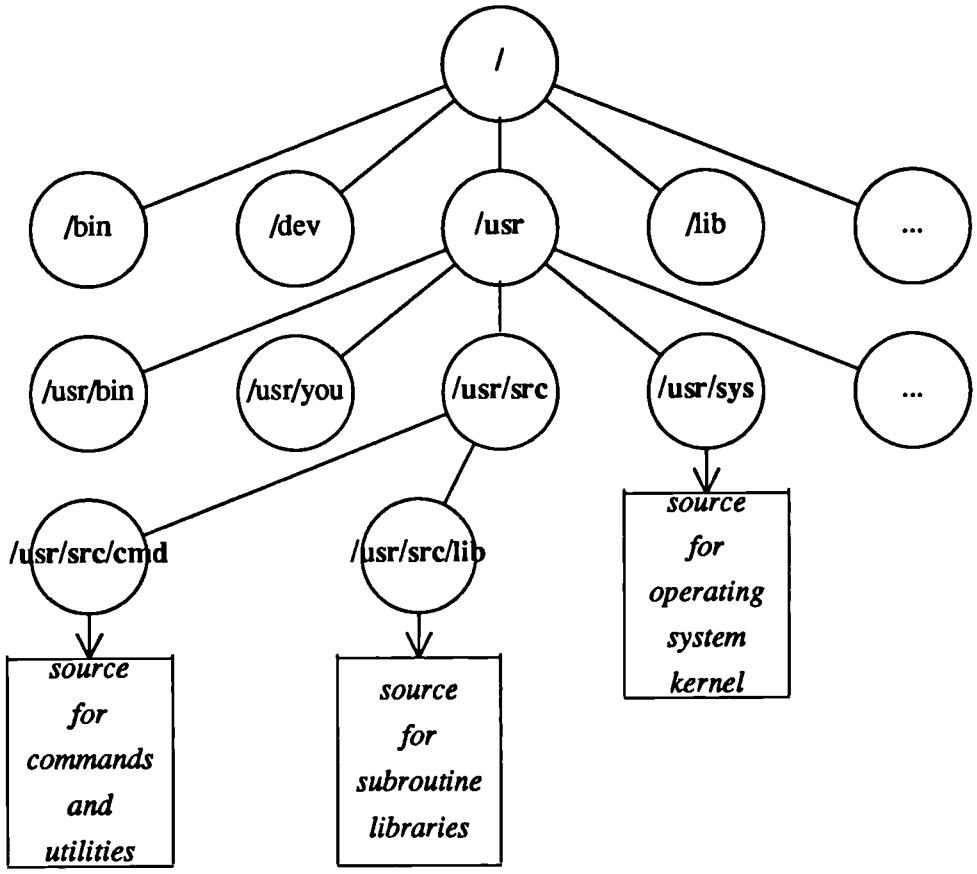
The CB/UNIX 2.0 and 2.1 tapes that were obtained were written using CB/UNIX's own, undocumented, archiving programs which this investigation could, unfortunately, not get hold of. Hence, their source trees could not be reconstructed. Finally, the PWB/UNIX 2.1 tape had some error which prevented a complete read.

4.5 SOURCE CODE STRUCTURE

Before discussing how the specific metrics were measured from the collected distribution tapes, lets look at how the source code tree is usually structured in UNIX. This section aslo describes the typical 'C' program structure (since most of UNIX is written in 'C' [JOH78]) and how this effects metric calculation.

4.5.1 UNIX directory hierarchy

From the earliest days, the source code was provided on-line in UNIX systems.[5] Indeed, this was one the main reasons for the initial popularity of UNIX, since it allowed other groups to easily tailor the systems to meet their own requirements. The hierarchical file system in UNIX allows the source to be structured in the following way:



The organization presented above is from the 8th Edition Research System and gives just a flavour of the directory hierarchy, for details see [KER84]. Most of the source is contained in the directories highlighted. Some of the other interesting directories are:

5. This practise has only recently been changed to enable the commercial groups to generate more revenue by charging extra for source licenses.

/	root of the file system
/bin	essential program executables ("binaries")
/dev	device files
/usr	the user file system
/lib	essential libraries
/usr/bin	less essential or user binaries
/usr/you	user *you*'s login directory

Earlier releases and releases from other groups have slightly different organisations and these are discussed in the *Changes* section, below.

4.5.2 The 'C' language peculiarities

Since most of the UNIX system is written in 'C' and the bulk of the analysis is centred on the 'C' proportion of UNIX, it is worthwhile spending a few moments to go through some of the characteristics of the language (a detailed description may be found in [KER78]). This will help decide which of the metrics described earlier in this chapter are best suited to measuring different aspects of programs written in 'C'.

Language features

C, developed by Dennis Ritchie in the early 1970s, is based on the language **B** which was written by Ken Thompson in 1970 for the UNIX system. Since B was based on BCPL, many features of C have roots in BCPL, as described below:

- The primary data types are characters, integers and floating point numbers. Other data types may be created by using pointers, arrays, structures, unions and functions.
- C provides the control flow structures required by "structured programming": statement grouping(`{}`); decision making (`if-else`); looping (`while, for, do`) and selecting one from a set of possible cases (`switch`).
- There are pointers and the ability to do address arithmetic. Function arguments are passed by value, though the effect of "call by reference" may be achieved by passing a pointer, allowing the function to change the object pointed to.
- Function definitions can not be nested, though they may be called recursively. Variable may be declared in block structured fashion and local (to function) variables can be

automatic or static.

- Functions may be compiled separately. Variables may be internal to a function, external but known only within a single source file or completely global.

Preprocessor features

C also provides some language extensions by means of a macro preprocessor. The extensions are:

File Inclusion

The contents of a file may be included in another file by having a line of the form:

`#include "`*filename*`"`

The preprocessor replaces such a line by contents of the file *filename.*

Macro Substitution

The preprocessor allows substituting a string of characters for a name, e.g.

```
#define    YES    1
```

changes all occurrences of `YES`s to `1`s in the rest of the source file which contains this line. This allows a way of defining constants which isn't there is the language proper (as there is in, for instance, Pascal). Such definitions can also be made from the command line at compile time.

Macros with Arguments

Macros may also have arguments to change the replacement text according to the way the macro is called, e.g.:

```
#define    max(A, B)        ((A) > (B) ? (A) : (B))
```

provides a "function" that calculates the maximum for any data type. However, since it is purely textual substitution, there are some problems, e.g.

```
x = max(i++, j++);
```

wll be replaced by

```
x = ((i++) > (j++) ? (i++) : (j++));
```

which will incorrectly increment `i` twice. This facility is

commonly used to avoid the overhead of function calls for 'simple' 'functions' and hence increase efficiency.

Conditional Compilation

The preprocessor also allows code to be conditionally compiled, e.g. the lines:

```
#ifdef VAX
      string1 = string2;
#else
      strcpy(string1,string2);
#endif
```

will be replaced by

```
string1 = string2;
```

if the identifier `VAX` has been the subject of a `#define` control line and

```
strcpy(string1,string2)
```

if it has not. This is most frequently used for portability, when the same source file is used on different machines but there are slight variations in the compilers requiring slightly different code. Conditional compilation is also used when the same source file is used in different programs and when the majority of the code in the source file is used in both programs.

Since the preprocessor processes the text in a separate pass before the compiler proper, the text seen by the compiler sees is very different from that seen by the programmer.

4.5.3 Program structure

Some of the features discussed in the previous section encourage programmers to adopt a particular structure style for their C programs.

- The facility of separate compilation encourages programmers to split the program up into several source files, keeping functions that logically belong together in the same file.

- Since separately compiled files may use common structures, definitions and global variables, the file inclusion facility allows these definitions, declarations etc. to be put in a separate file (or files if they are many) which is included in the main source files. This

prevents repeating the definitions in each file with all the associated problems. Traditionally the files which contain these definitions etc. are called headers and have the suffix .h while main source files have the suffix .c (this is required by the compiler).

In addition, most UNIX systems provide 'standard' header files which contain definitions required by standard libraries.

4.5.4 Program Construction

With several source files and header, possibly in different directories, constructing programs can become quite difficult. To cope with this most 'C' programmers (and certainly distribution tapes) use the *make* utility [FEL79], to keep track of which files depend on which others and to handle the compilation. *Makefiles* are written which contain the dependency trees and rules to compile the files and the executables.

Often, the search paths for the *include* files and the some *#define*s are also given in the makefiles.

4.6 MEASUREMENT OF THE METRICS

This section describes how the metrics described earlier in this chapter were measured in this project. Since the support documentation in UNIX groups was very weak, this study had to rely on the on-line source code and manual pages for almost all the measurements described below.

4.6.1 Preprocessing

A prerequisite for processing the source files in any way, to get to the required metric measure, is to identify the source files on the distribution tape. Most of the more sophisticated metrics require the code to be parsed in some way. The techniques to perform these two actions are described below.

Finding source files

The hierarchical file system in UNIX makes it simple to locate all the 'C' source files: we simply traverse the whole tree looking for files which have the suffixes *.c* or *.h*. The following invocation of the standard UNIX utility find prints out all the required files.

```
find / -name '*.[ch]' -type f -print
```

There is one drawback with this approach in that *all* files with names ending in .c or .h will be counted, even if they are not 'C' source files since UNIX does not have file types to separate 'C' source. However, since the main sources of data are distribution tapes, this strategy is safe in that very few (none that this investigation came across) standard non-C files are likely to have suffices

.c or .h.

The releases for which private copies were used (because distribution tapes were not traceable) had to be massaged to remove non-standard files. This was done manually and involved some guessing, so analyses of those releases are necessarily less accurate than the analyses derived from distribution tapes.

Parsing

Since compilers have to parse the source before they can generate the assembler, it seems reasonable to modify the 'C' compiler, slightly, to perform the desired analysis and this has been done in the past, e.g [BEN79].

Since this study had access to the source of the 'C' compiler at Bell Labs., where most of the analysis was conducted, the compiler was modified to keep a count of the various tokens it recognised. The rationale behind this was that most metrics are dependent on counts of various tokens (e.g. McCabe: decisions; Halstead: operators and operands) so if all of them were counted individually, any metric could easily be constructed. The original intention was to use just the parsing part of the compiler and discard the rest, but this was not fulfilled because the parsing code could not be cleanly separated. Hence, the 'tool' was extremely inefficient since it spent most of its time compiling the 'C' source file, the result of which was thrown away! The modified source of the compiler and the shell script used to drive it are given in Appendix D.

This approach, of using a modified compiler to analyse the source code, is reasonable because source files supplied on distribution tapes are expected to at least compile successfully. However, this was not the case in the analysis of UNIX systems, due to several reasons:

- UNIX has gone through several machine changes in the period for which there is data (1975-1986) To keep up with this, the 'C' compiler has also changed. Therefore some of the older files could not be compiled successfully (i.e. presumably, they would have compiled on older compilers, e.g. for the PDP-11, but did not on the VAX-11/750 where most of the analysis was done).

- Sometimes include files could not be found. As most of the analysis was done on a machine which was in constant use by others, the releases to be analysed were mounted in a directory which was not root (where they expected to be mounted). Hence references (e.g. for *include* files) to absolute pathnames were incorrect.[6]

- Sometimes the wrong bits of code were chosen when conditional compilation code existed, because the flags used to drive the *#ifdef*s were not set correctly in the code.[7]

4.6.2 Size

The size metrics can be divided into two classes for measurement purposes: those that require the input (i.e. the source code) to be parsed in some way, and those that don't. Since a parser was available, all the metrics in the first category were measured by processing the output from the parser and are subject to the limitations described in the previous section.

To get a measure for the whole distribution, the individual measures for each file were simply added up, using the standard UNIX utility awk. The awk scripts are given in Appendix D.

Size metrics requiring parsing

The output of the parser gave us counts of various tokens, including:

1) individual counts for each operator (+,-,/,*, etc.)
2) individual counts for each type of statement (if, while, for, etc.)
3) variable and total operand occurrences
4) function calls and external (to file) function calls
5) external variables and variable (internal to function) definitions
6) number (and size in characters) of comments
7) function definitions and function parameters

With these in hand, it was relatively straight forward to measure most of the metrics:

Number of statements	Simply add up the individual counts in (2) above.
Halstead's	Number of unique operators = number of individual counts with non-zero values. Number of operator occurrences = total of (1) above. Number of distinct operands = sum of (7) and (5) above.

6. As an aside: research systems tended to use relative pathnames and so could be compiled, while USG, UDL and ATT-IS systems tended to use absolute pathnames which made them much less compilable.

7. In practise, they were set in *makefiles* which were not used the analysis. Using the supplied makefiles would have further complicated the analysis, primarily because this investigation was using a shared machine while the supplied makefiles assumed that the distribution was being to construct the host UNIX system. It would be interesting for a future project to use a dedicated machine to analyse the sources, which would allow much more flexibility.

	Number of operand occurrences from (3) above. All of Halsteads size metrics are based on the above measures.
Number of functions	The first count in (7).

Since the parser used the 'C'-preprocessor, the source was processed after the inclusion of the include files and hence the counts include them and are therefore much larger than what a programmer would see looking at a listing of the source file. Also, since the counts are after macro expansion, some "functions" may not be counted as functions.

Size metrics not requiring detailed parsing

Since the following measures were obtained without compiling the code or passing it through the 'C' preprocessor, the count does not include the include files. It does, however, include all the preprocessor directives in the file. Hence, they are very close to what the programmer would actually see, if he were looking at the code.

Lines of code	Use the standard UNIX utility wc to count the number of lines in each file. Note this includes all lines, including blank and comments.
Number of characters	Use the standard UNIX utility wc to count the number of characters in each file, including comments.
Number of files	Simply count the number of files in the file list.

Others

It was not possible to measure the following size metrics:

1) lines which are not blank and not comments
2) lines which contain executable statements

4.6.3 Complexity

Since most of these metrics also required parsing the code, they were also derived from the output of the parser.

McCabe's	Totaling the number of *if* and *case* statements in (2) above.
Halstead's	They are constructed from the four basic measures mentioned in the previous section.

Number of external links	By counting the number of global functions and non-static external variables.

It was not possible to measure the other complexity measures described in the earlier parts of this chapter.

4.6.4 Work-rate

Since information for the inter-release period was not available, the amount of work done in that period could not be measured directly. However, by counting the number of files changed, added or deleted in the interval, an indication of the effective work-rate was obtained.

Measuring which file has changed is determined by checking the size (in characters) in successive releases, if the size is the same then the file is counted as not having changed. This technique is necessitated since, because of the lack of information, there is no firm way of knowing which files have changed. While it is possible for a file to have changed without changing its size, it is extremely unlikely. If a filename existed in the old release but not the new one it was deemed deleted, if it existed in the new one but not the old one it was deemed added. The **awk** script to do this is attached in Appendix D.

There was another problem which had to solved before accurate measures of files changed could be obtained. During the life-time of a particular branch of UNIX, some general directory hierarchy changes were conducted which effected the source tree. In effect, the file's name was changed. In a simple strategy, this would count as one deleted and one new file, whereas it is the same file. For instance: in the earlier research releases the source for the kernel was stored in the the directory */usr/sys/ken*, but in later releases, the kernel is stored in */usr/sys/sys*. After experimenting with automatic ways of determining name changes[8] and failing to detect all known cases, it was decided to do map the changes manually. All the file names in the earlier releases were changed to reflect the convention in later releases. This was done for all three releases.

The metric presented in the graphs (in the next chapter) is: number of files changed, deleted and added divided by the release interval in months. This is equivalent to the "number of modules handled" metric used previously.

---

8. By comparing only the last two (then three) components of the filename, instead of all of it.

4.6.5 Increased Complexity

The complexity metrics presented in the previous section give a measure for a single piece of code. That is, given any piece of code, the preceding metrics will come up with a figure.

To measure a *change* in complexity, Lehman used a metric which measured the impact of the new release on the old release. The rationale being that if more of the existing system needed to be altered, the system was becoming more complex. This metric has the drawback that it relies on the release content being constant (which may not be the case). Furthermore it has been shown to be susceptible to large release intervals [KIT82].

The measure is number of files changed or deleted in the new release divided by the total number of files in the release.

4.6.6 Release Content

Kitchenham's metric could not be used, for any branch of UNIX, since there was insufficient information available to decide how many versions a particular file had gone through. Lehman's metric is simply net growth, which is the difference in size between the two releases and this was easily calculated.

4.6.7 Others

The lack of the necessary information made the measurement of other metrics impossible.

4.7 CHOICE FOR UNIX MODELS

It was decided to stick to Lehman's metrics for the required attributes because they were the ones with the most accurate measures.

The *lines of code* metric for size, was another accurate measure but it was found to give no more information than the *number of files* metric.

There were severe problems with metrics using the parser, as described earlier. It was found that between 10% and 30% of the files on the distribution tapes could not be parsed.

Even for the files that could be parsed, neither Halstead's metrics nor McCabe's nor any of the other count based metrics gave any more information than a *number of statements* metric.

In any case, because the *include* files had to be measured in every file analysed, hence repeating their count several times, the metrics which required source parsing were not thought appropriate to model 'C' source code.

The plots of all the metrics rejected here can be found in Appendix C.

Chapter 5
MODELS OF THE UNIX EVOLUTION PROCESS

"By a small sample we may judge of the whole piece."
(CERVANTES, *Don Quixote*, 1605-15)

5.1 INTRODUCTION

By looking at plots of various system and process attributes, this chapter examines the UNIX Evolution Process and attempts a prognosis for its future. Hence, it critically evaluates Lehman's concepts of Program Evolution on which the statistical modelling techniques used here are based.

The UNIX Evolution Tree

The history of UNIX has already been described in detail in *Chapter 3*. This section of the thesis will attempt to model the evolution of three of the most well established streams of UNIX still surviving today. They are the UNIX releases from (in chronological order):

- The Computing Science Research Center at Bell Labs, Murray Hill.
- The UNIX Support Group (USG), which grew into the UNIX Development Laboratory (UDL) and eventually the Software Division of AT&T Information Systems (ATT-IS).
- Computer Systems Research Group in the Computer Science Department in the University of California at Berkeley.

It turns out that the above groups represent three different programming cultures (i.e. research, supported/commercial and academic) and therefore create an ideal opportunity to examine the effects of the cultural differences on the dynamics of programming process.

Data Base for the Plots

Due to the very poor record keeping at all the UNIX centres (documented in Chapter 4), a complete database of all the relevant releases could not be constructed for this project. Therefore the plots presented in this chapter are restricted to information derived from the the following releases:

Research	Editions 5-9 denoted as v5-v9
Supported/Commercial	Generic 3.0 (denoted as g3), UNIX/TS 1.0 (ts1), Releases 3.0, 3.0.1, 4.0, 4.2 and 4.2.1 (denoted as r3.0 etc.), System III, System V releases 1 (denoted s5), 1.1, 1.2, 2.0, 2.2 and 3.0 (denoted s5r3 etc.).
Academic	BSD/UNIX 3.0, 4.0, 4.1, 4.2 and 4.3.[1]

The above represents an almost complete sequence of the relevant releases from 1975, 1977 and 1980 respectively. The source information extracted from these releases is supplemented by release information from other records.

Statistical Interpretation

Most of the graphs are scatter plots of the data. Where more than one set of data is presented in the same graph (e.g. the size vs time plot), points belonging to the same set are connected with straight lines. Tests for linearity are done by applying least squares fits to the data.

5.2 THE PLOTS

This section presents the plots which show the evolution of the pertinent system/process attributes for the three branches of UNIX described above. The idea is to get a feel for the way UNIX has evolved, particularly the difference between the three branches, and how close the shape of the evolution pattern is to that predicted by Lehman (as described in *Chapter 2*).

Structure of this section

Each subsection is devoted to a particular attribute and begins by explaining how the attribute was measured. Then plots of its value in successive releases from the three streams are presented.

1. These are *all* the BSD/UNIX releases. Remember that BSD 1.0 and 2.0 were not complete systems, just collections of software to *run* on UNIX (see Chapter 3).

The fourth plot shows the value of the attribute at different points in time (plotted on release dates). The accompanying text discusses any points worthy of note, particularly their similarities to the patterns predicted or expected by Lehman's "Theory of Program Evolution".

Graph Axes

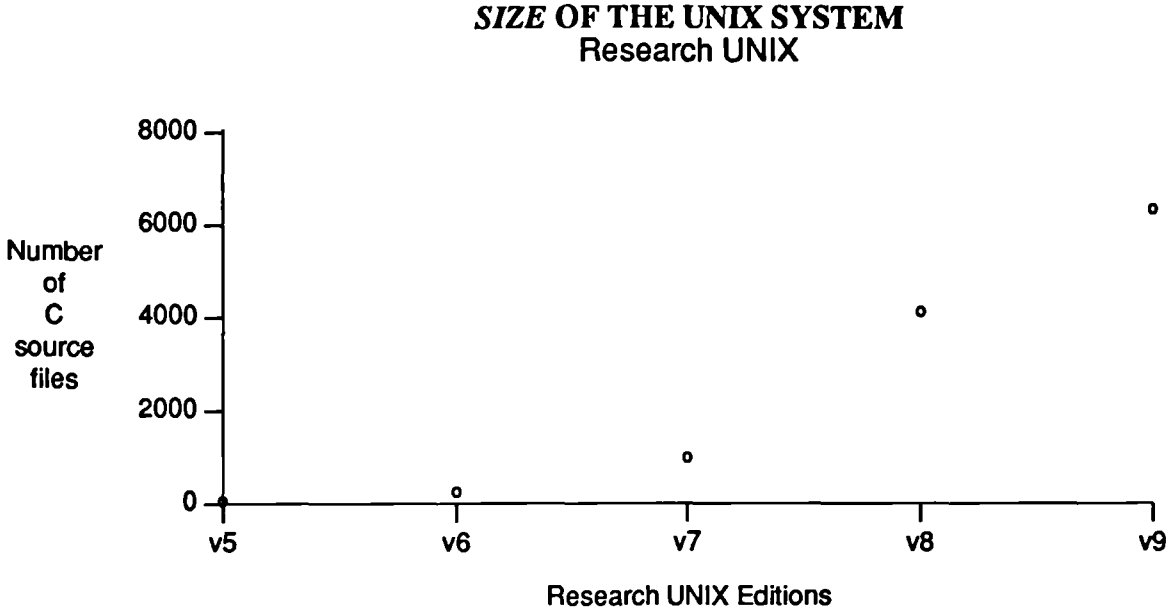
So as not to disturb the reader's visual perspective, the Y-axis (giving the attribute value) scale is not altered in each set of graphs (corresponding to a subsection). The X-axis on the first three graphs is the Release Sequence Number (RSN) for the three branches (e.g. v1-v9 for Research). The final evolution[2] graph shows the value of the attribute (for all three branches) as a function of calendar time, the three streams are shown as three distinct lines as follows:

- Research: dotted line.
- Academic: dashed line.
- Supported/Commercial: solid line.

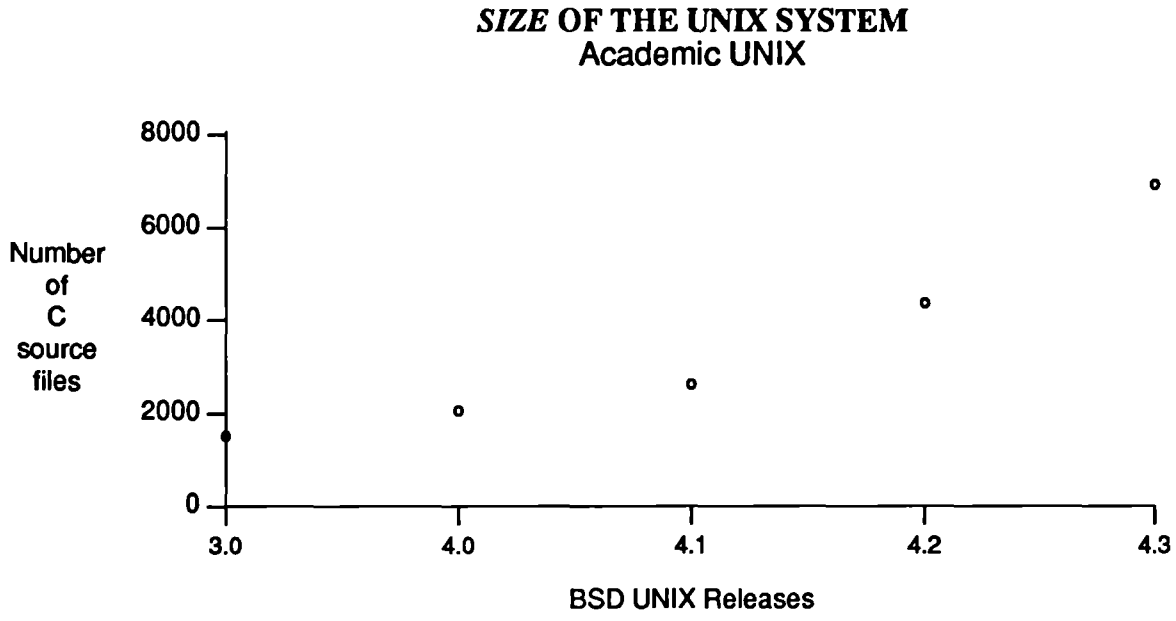
5.2.1 Size

This subsection shows how much bulk the UNIX systems have accumulated over the years. The size of the system, as shown here, is a count of all the 'C' source files (including headers - see explanation in Chapter 4). In 'C', this is equivalent to the "number of modules" measure used in previous studies of this kind (e.g. [CHO81]). One more point worth mentioning here is that this measure counts *all* the source files on the distribution tape, this includes all the utilities and the applications supplied with the system, and not just the kernel.

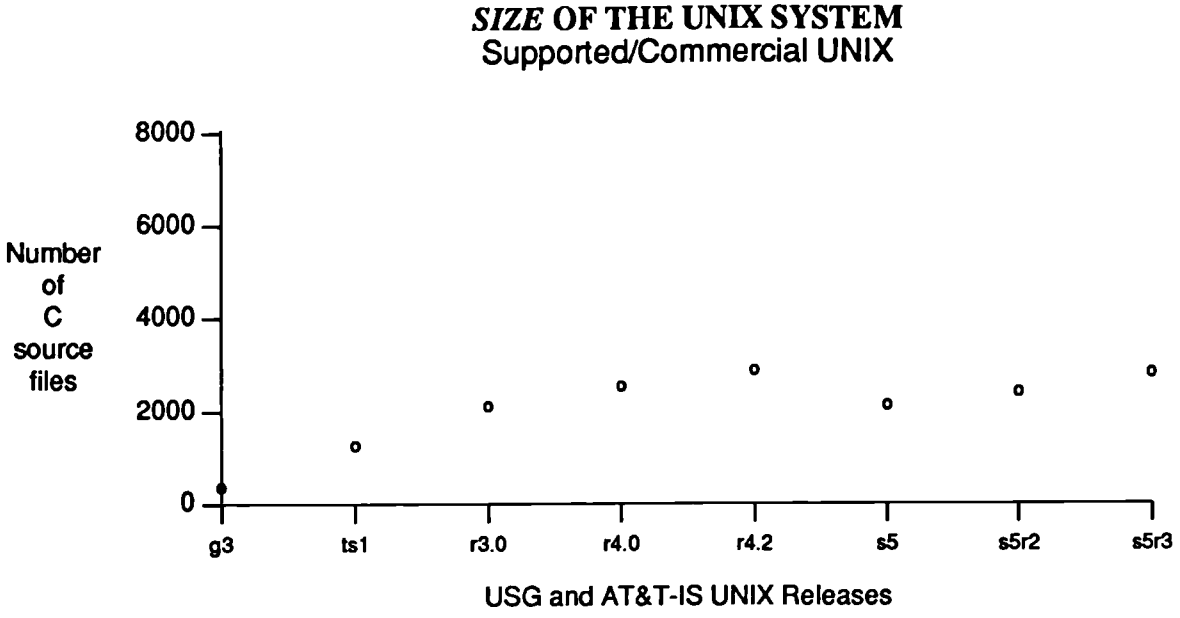
2. For there may be least squares fit plots.



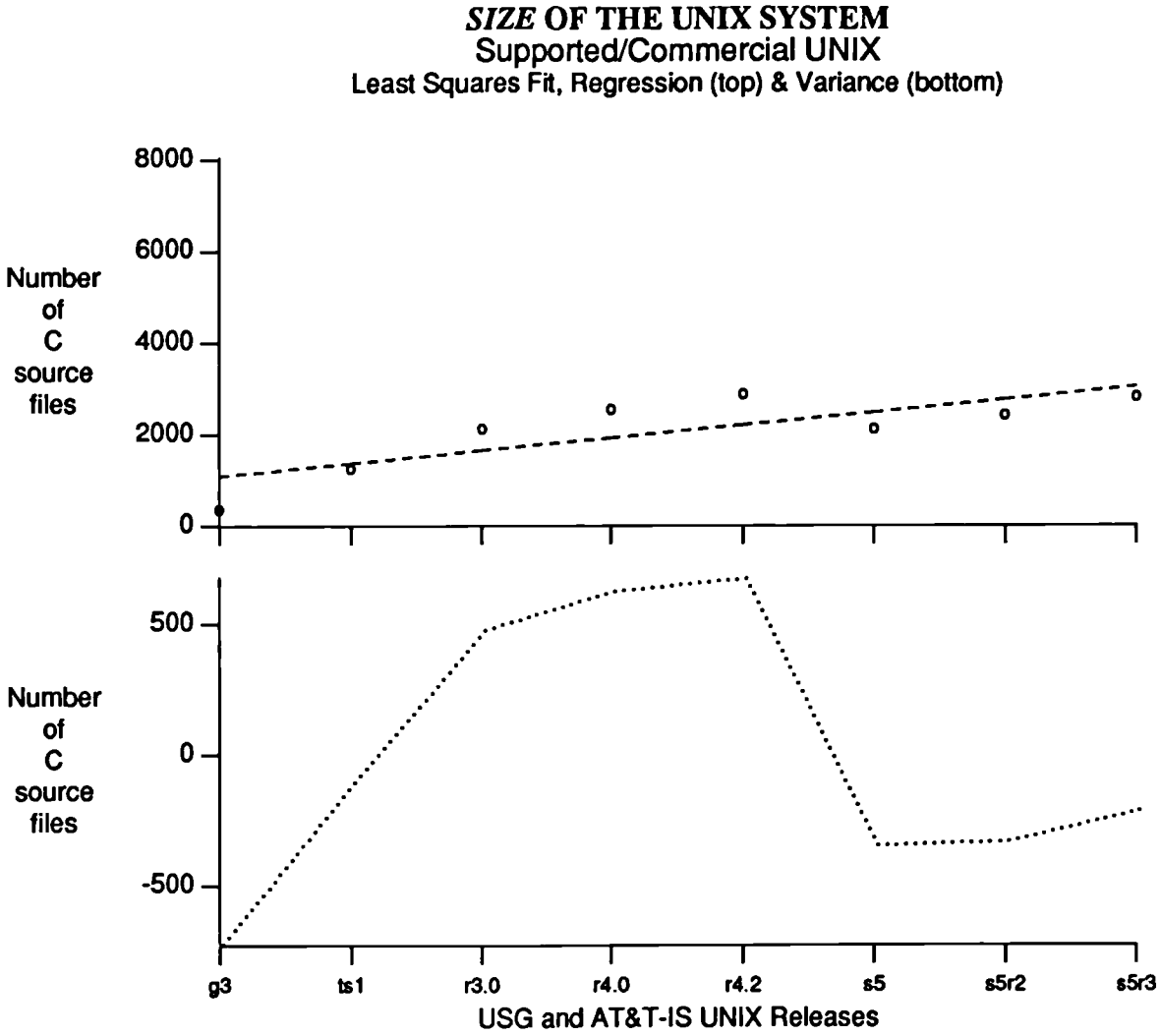
The graph shows that Research UNIX grew steadily but slowly until v7, then much more rapidly. There are two immediate explanations for this. One is that v5-v7 were for the PDP-11 processors while v8 and v9 are for the larger VAX processors. The second, and probably more likely (since other porting exercises [LON78] [JOH78] of UNIX did *not* result in large code increases) explanation is that v8 is based as much on 4.1 BSD as on v7. These BSD acquisitions (the size of 4.1 BSD was about 3,000 files at the time of transfer to v8) are probably responsible for the swelling in v8. The large increase in size between v8 and v9 is explained in the 'GROWTH' subsection.



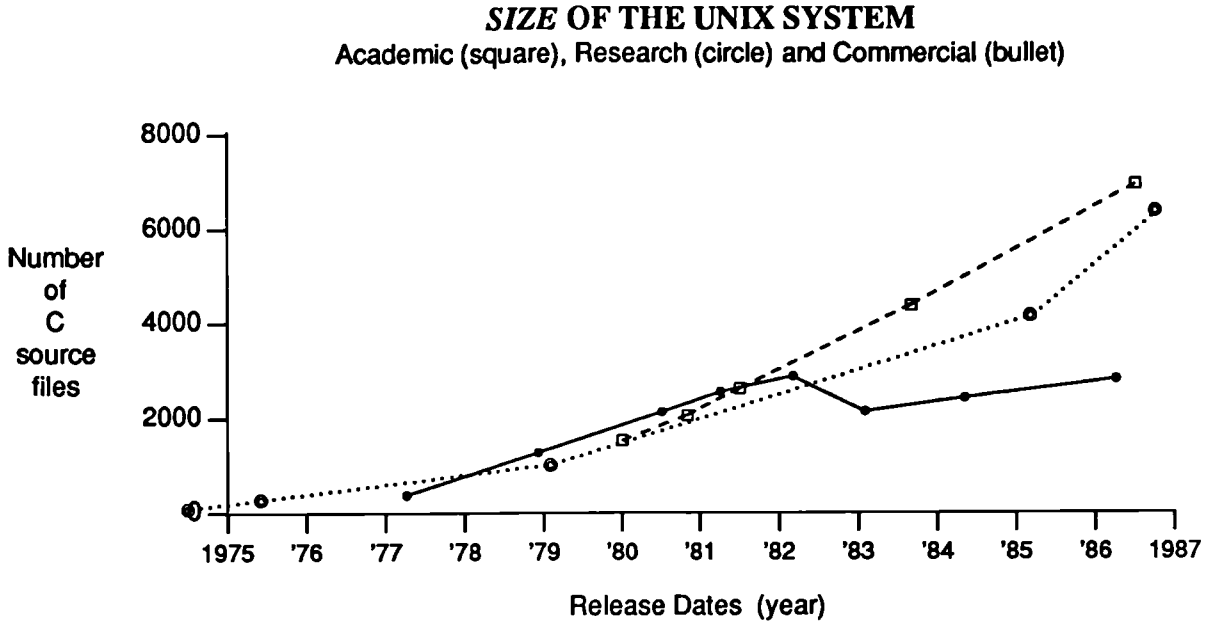
Again there is a steady growth pattern except that it seems to be smoother (in the sense that there isn't a 'step' as there was in Research at v7). The large increase in size at the release of 4.2 BSD is understandable in the light of the new features introduced by the release, including a new file system, new networking and IPC facilities and virtual memory. However, the large growth between 4.2 and 4.3 is unexpected since 4.3 has been regarded as a clean-up release and has not introduced any significant features.



There seem to be two main stages in the growth of supported UNIX: the first ending at the release of 4.2 and the second starting with the release of System V. On slightly closer examination, a slight step is found at Release 3.0 (where the slope seems to decrease slightly). There are not enough points to confirm any of these observations but since the three stages visible on the grap tie in with milestones in the evolution of supported UNIX (Release 3.0 was the first release from the consolidated Unix Development Laboratory and System V was the first commercial release of UNIX), the stages theory seems at least plausible. Furthermore, the slope of the graph from s5-s5r3 seems strikingly similar to that between r3.0-r4.2 (again there aren't enough points to test for this).



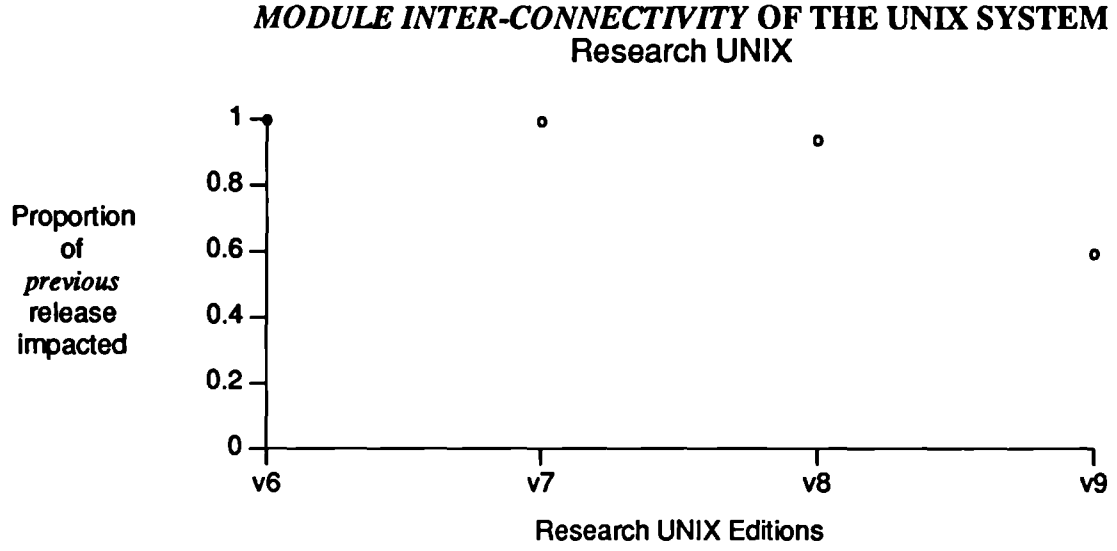
The least squares fit shows a cyclic fluctuation, with reducing deviation, around the regression line. This is consistent with Lehman's hypothesis of the growth of software systems. Indeed, it can be said, based on past experience, that commercial UNIX will continue to grow at *280* files per release.



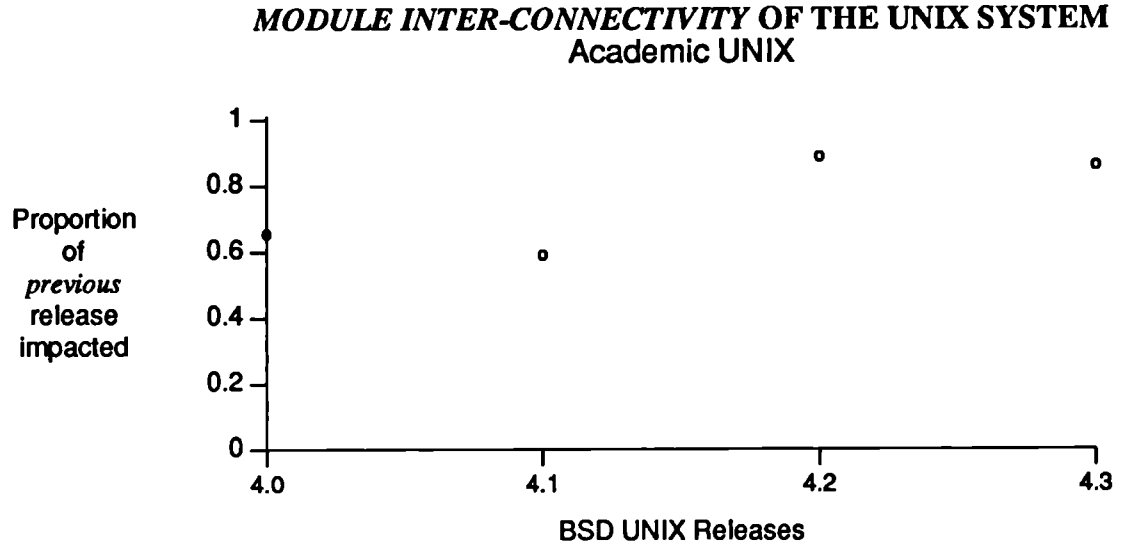
When the sizes of the UNIX systems are plotted against time, the size of BSD/UNIX appears to be increasing linearly. Its growth rate of roughly *850* files per month seems to be higher than the other two. Apart from being monotonically increasing there is no visible trend for Research UNIX. The three stages that were visible in the previous plot have been reduced to two in this one. The growth seems linear until 1982 and then resumes linearity after 1983 *but with reduced slope* (*510* vs *215*). This is the only indication that there has (possibly) been an increase in the level of difficulty in working on commercial UNIX. It is interesting to note that the point at which this increased complexity becomes visible is the commercialisation of UNIX. Surprisingly, supported UNIX seems to be about half the size of BSD UNIX which is only slightly bigger than Research UNIX, yet around 1981 they were about the same size!

5.2.2 Module Inter-connectivity

This subsection views the evolution of one measure of structural complexity in the three branches of UNIX. The metric measures how much of the previous release has been affected by work for the new release. The idea being that if new work requires an increasing proportion of old work to be changed, then the system is becoming more difficult to work with and complexity is increasing (this is discussed at length in Chapter 4). This metric, a measure of *module inter-connectivity*, is measured as number of previous release files changed or deleted in the new release divided by the size of the previous release. A figure of one implies the whole system being affected.

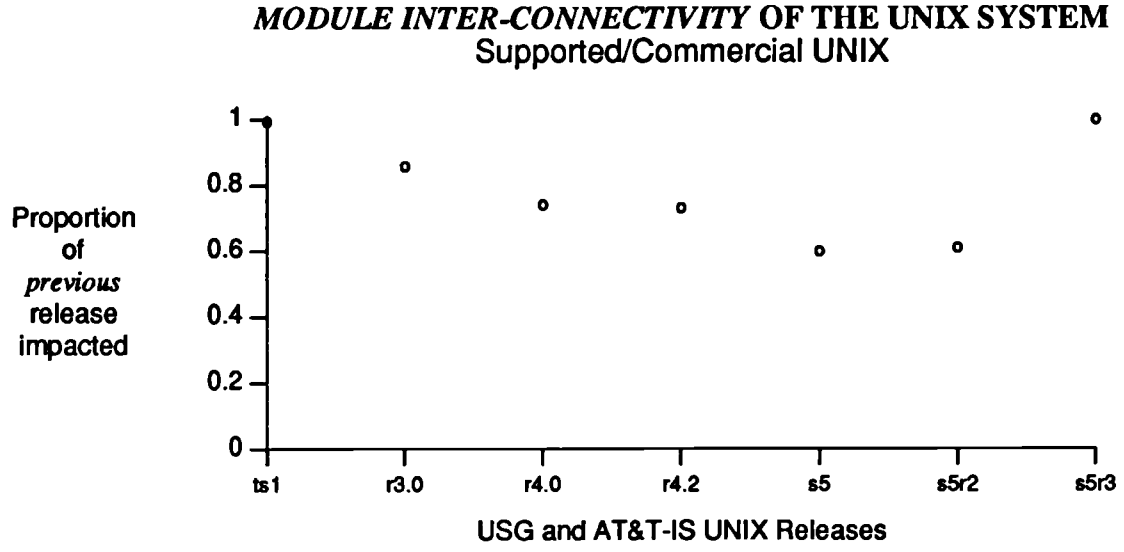


The graph shows previous releases being substantially affected by work for the new ones. However, a high figure is not necessarily a result of high complexity because given more time, other things being equal, the programming team will change more of the system and, as will be shown, the Research process has very long release intervals. What is encouraging is that is that the figure is decreasing, especially the drop between v7 and v8 which is in spite of the long interval implying that the cleaning up between v6 and v7 (which incidentally is responsible for *that* high figure) had been useful. The high figure between v5 and v6 is also not necessarily worrying since it is known (see Chapter 3) that all the files were changed to remove the copyright notices!

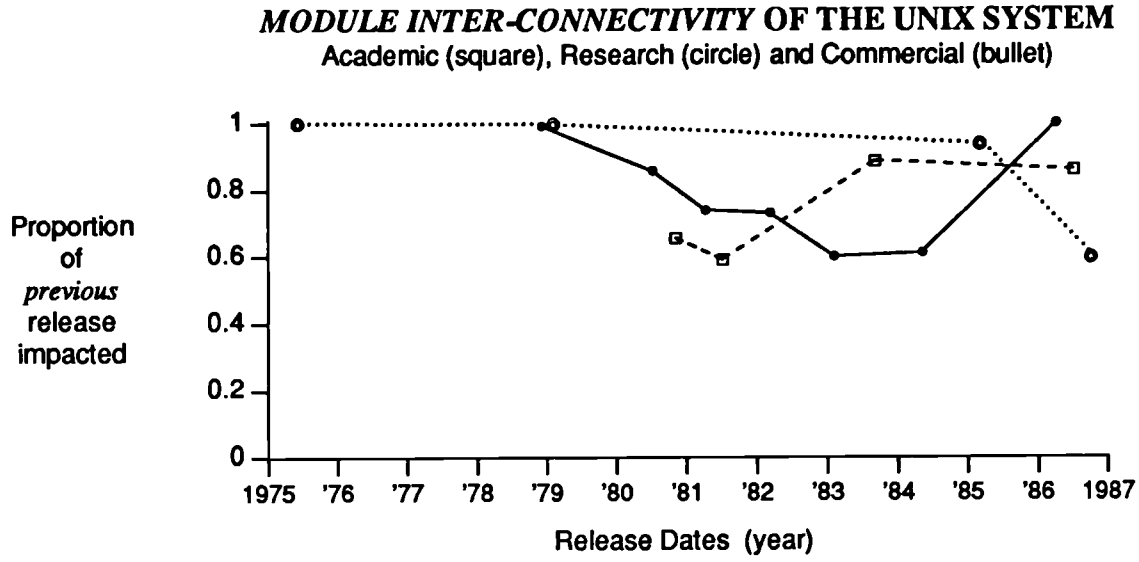


The graph displays a pattern reminiscent of OS/360 [LEH80] but with (as is the case with all BSD plots) too few points to test for a complex pattern with any certainty.[3] That trend was a mild zig zag, super-imposed on an overall increase. The peaks nicely coincide with the feature releases (4.0 & 4.2) while troughs with the clean-up ones (4.1 & 4.3). The upwards overall trend, perhaps, indicates increased complexity since more of 4.2 was changed by 4.3 than 4.0 by 4.1.

3. This will be the case throughout this chapter, whenever a pattern is suggested, it will be qualified by mentioning the small number of points.



Since the minor 'releases' have not been included in this analysis (they are the updates 3.0.1, 4.2.1, s5r1.1, s5r1.2 and s5r2.2 which affect a *very* small number of files and would only clutter up the plots), all the releases here are major releases so high-ish inter-connectivity figures would be expected, and this is borne out by the graph. Furthermore, the graph shows an overall decreasing trend until s5r2 indicating a *reduction* in complexity. But the sharp rise with s5r3 is worrying, since it is the first UNIX release from ATT-IS, the high complexity could be a result of changes in management policy, organization or simply commercial pressure, it is too soon to tell if this trend will persist.

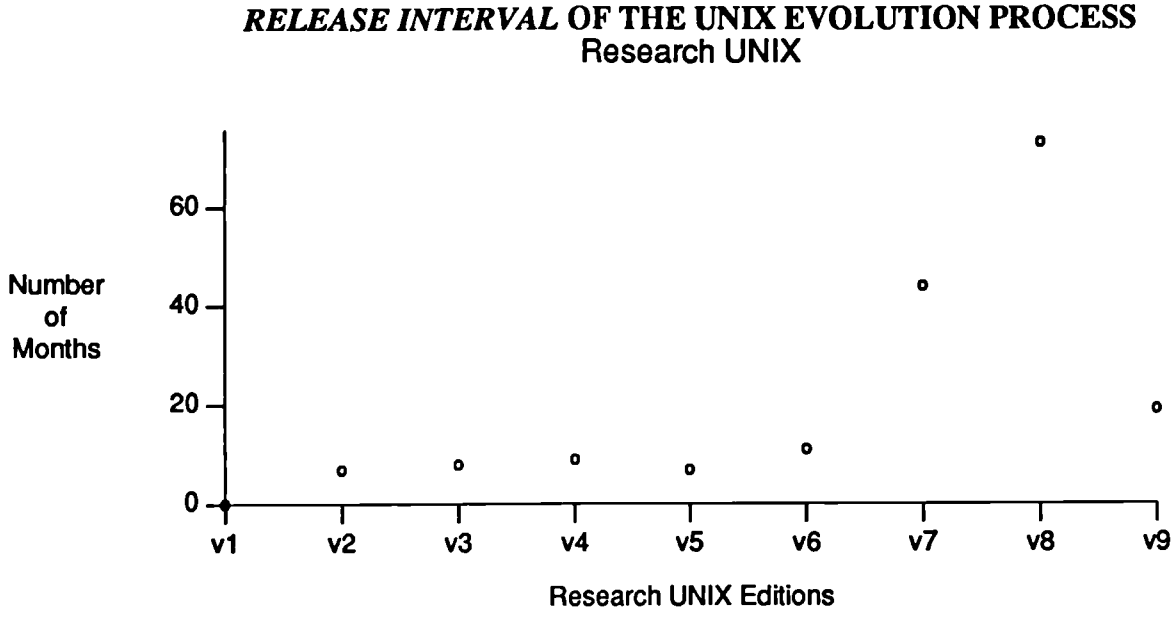


The time graph makes it easier to see the different behaviour of the processes but doesn't offer any additional insights.

5.2.3 Release Interval

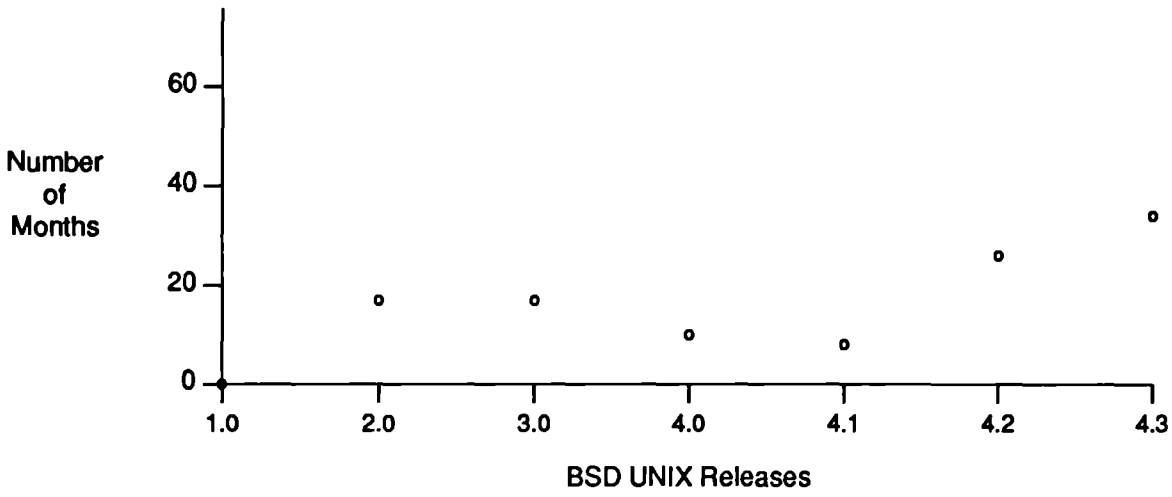
Another indication of increasing complexity is if the interval between successive releases goes up. The rationale behind this is that if it is taking more time to satisfy a similar requirement, then complexity must be increasing. The metric is measured in the time interval (in months) between successive releases.

Unlike other plots in this chapter, these plots are not derived from the scarce source code records and are more complete, hence they have more data points.



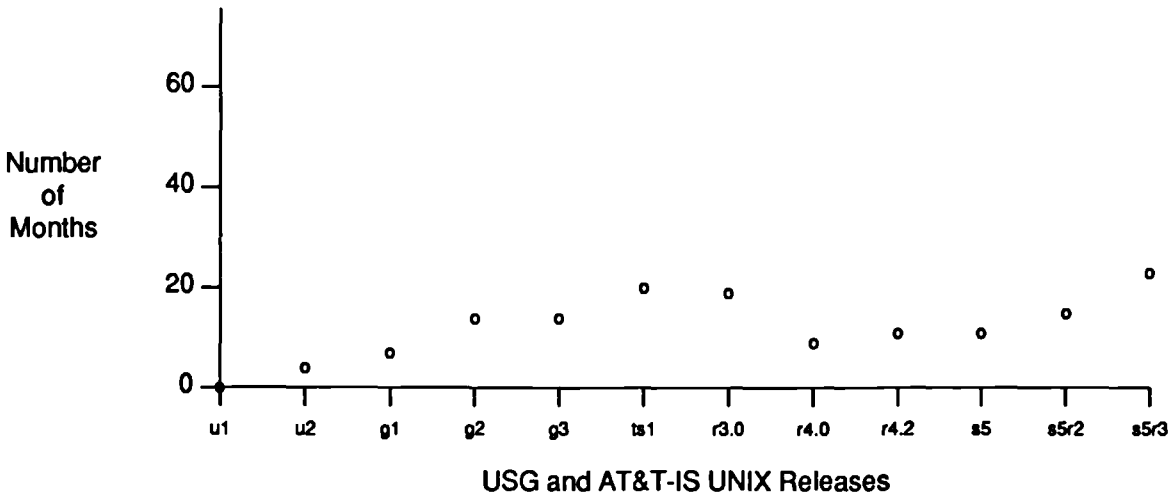
Before the release of v9, it could have been said that the release interval of the Research UNIX process in increasing steadily but with no discernible trend. As it is, with the short interval between v8 and v9, that can not be said. However, one possible hypothesis is that a new cycle has been started where a cycle consists of slowly increasing intervals followed by steeply increasing intervals. But there is not as yet enough data to test this.

RELEASE INTERVAL OF THE UNIX EVOLUTION PROCESS
Academic UNIX

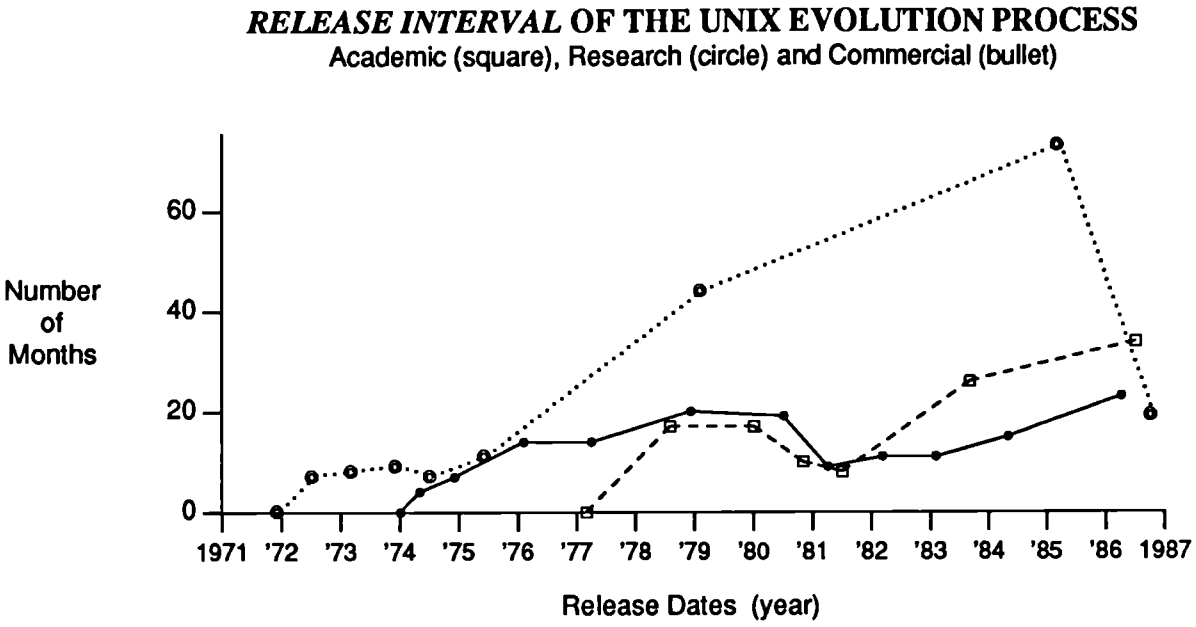


There is not enough data to draw any conclusions but the large rise after 4.1 BSD roughly resembles that of some other processes studied previously.

RELEASE INTERVAL OF THE UNIX EVOLUTION PROCESS
Supported/Commercial UNIX



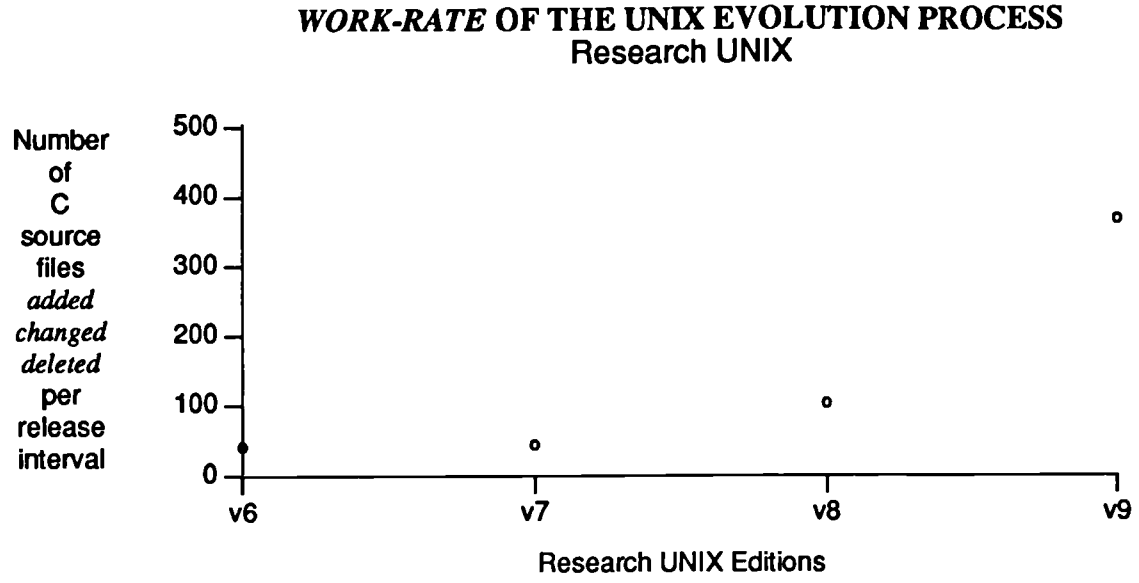
In this graph, two vague phases are visible. The first lasting until r3.0 and the second starting from r4.0. Both stages show rises but the second appears to be smoother (i.e. there aren't the flat spots of the first stage between g2 & g3, and between ts1 & r3.0) and closer to the pattern predicted by Lehman. There is no obvious explanation for the drop at r4.0 but it does support the cyclicity concept introduced earlier!



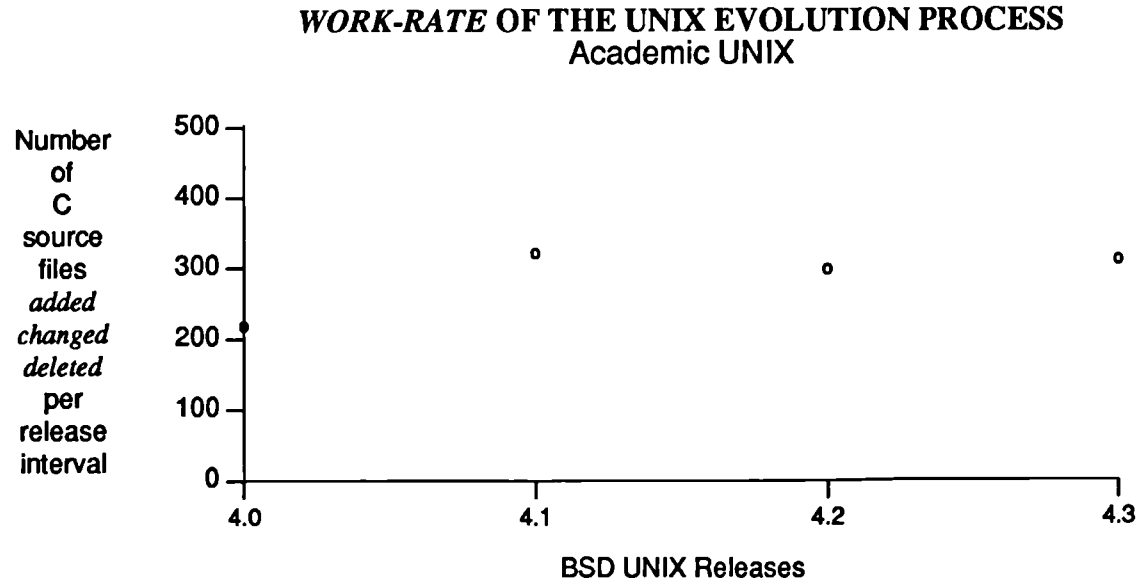
The time plot serves to highlight the dominating effect of the large release intervals between v6,v7 and v8 in the Research stream. The other interesting, though not especially useful, observation is that the value of the release interval before the rise in all three streams is around the 10 month mark. Furthermore they all seem to have the same slope in the rise portion of their graphs.

5.2.4 Work-rate

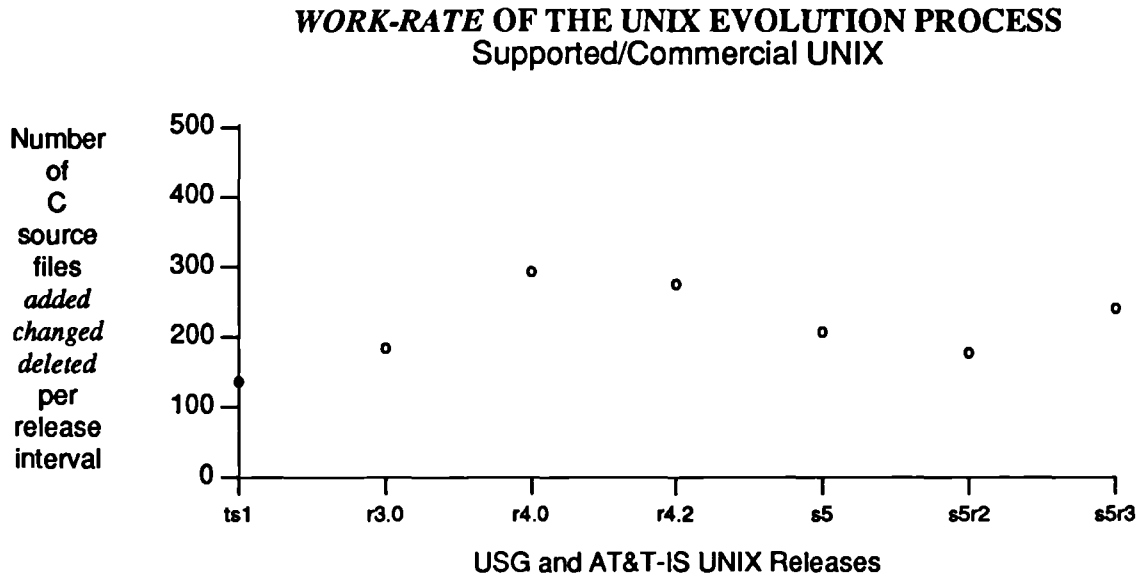
This section deals with the work-rates of the three UNIX processes under examination in this chapter. Work-rate is defined, here, as the total work done on the system in between two successive releases divided by the release interval. Since there is little direct information for the inter-release period, the releases themselves have to be interrogated to determine how much work was done for them. By counting the number of new files and previous release files changed or deleted in the new release, a feel for the amount of work done can be obtained (this is discussed at length in Chapter 4).



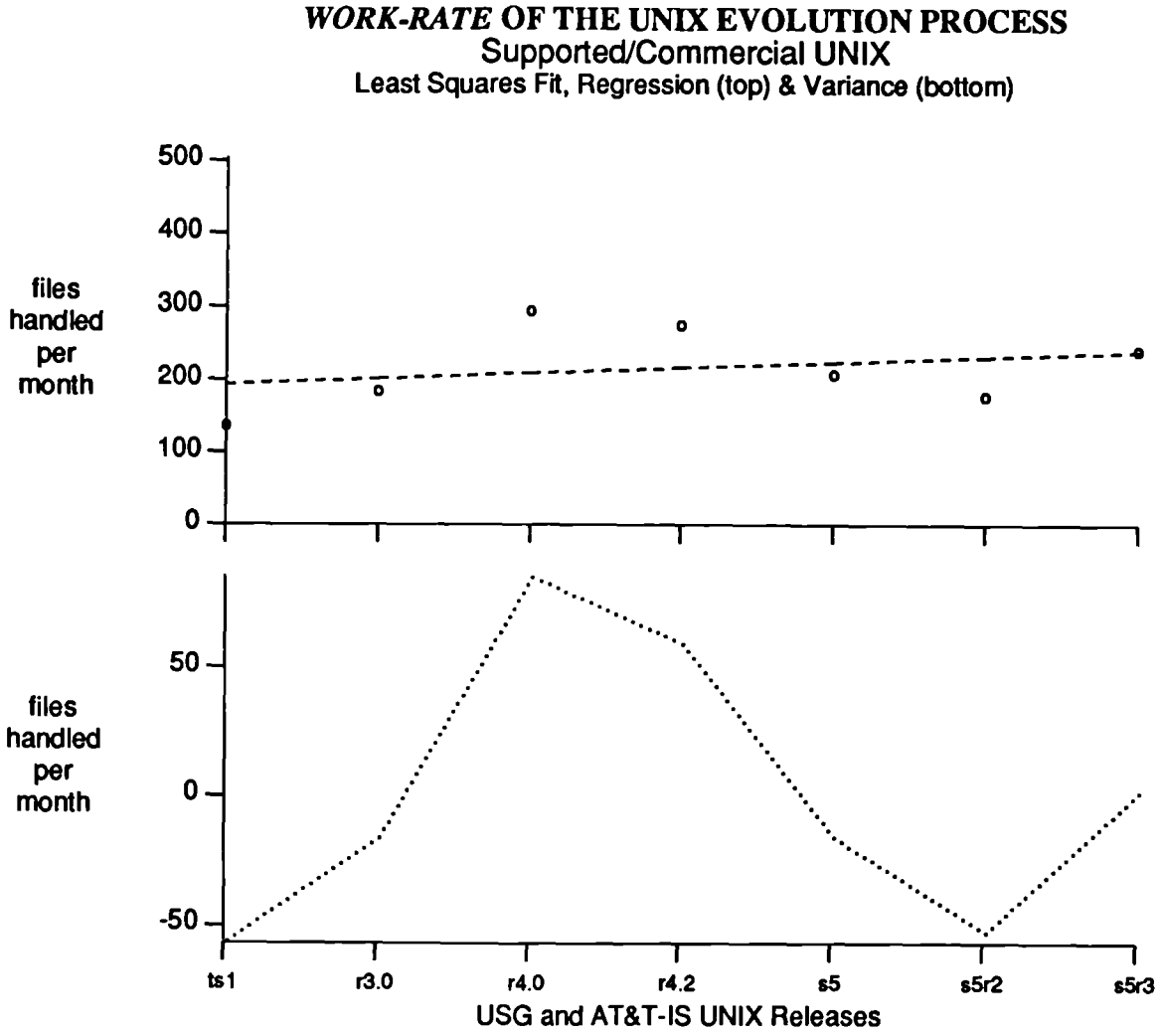
The relatively long intervals between v6,v7 and v8 have restricted the work-rate for the Research Process to below 100 files per month, until 1985. The increased effort to get v9 ready (in a short space of time) pushed the work-rate up dramatically indicating that the CSRC has a measure of control over the process but there isn't enough information to calculate an "average" trend.



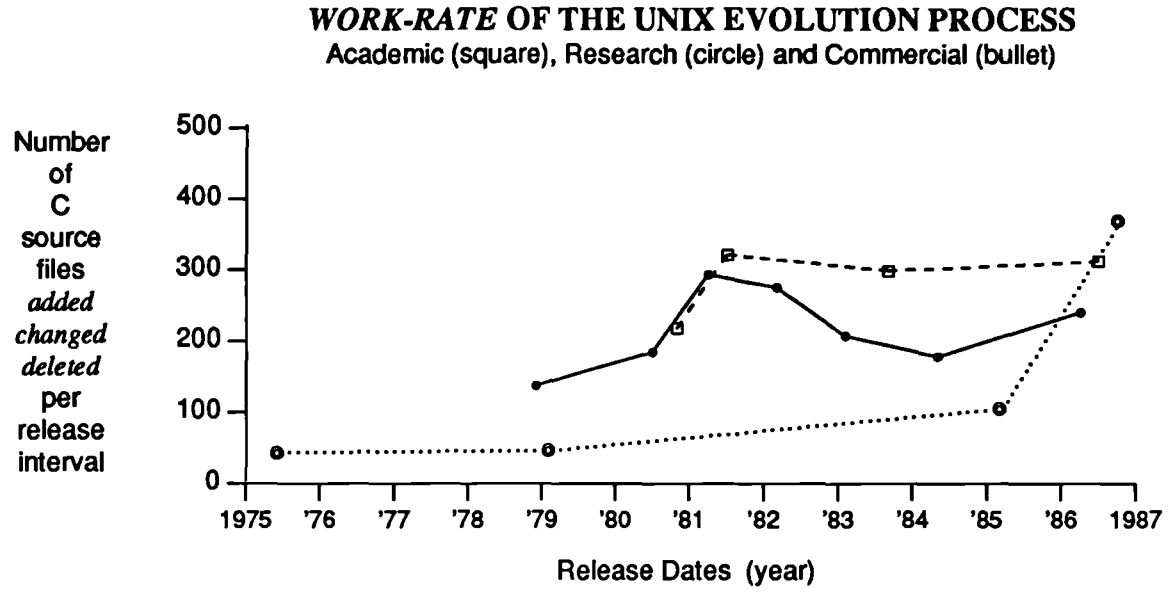
It appears that the increased backing of the BSD/UNIX effort (by DARPA and other sponsors - see Chapter 3) after 4.0 was reflected in an increase in the work-rate. It seems to have remained at that level since but it remains to be seen if this is a long term trend.



There is just enough of a visible trend here to attempt a least squares fit:



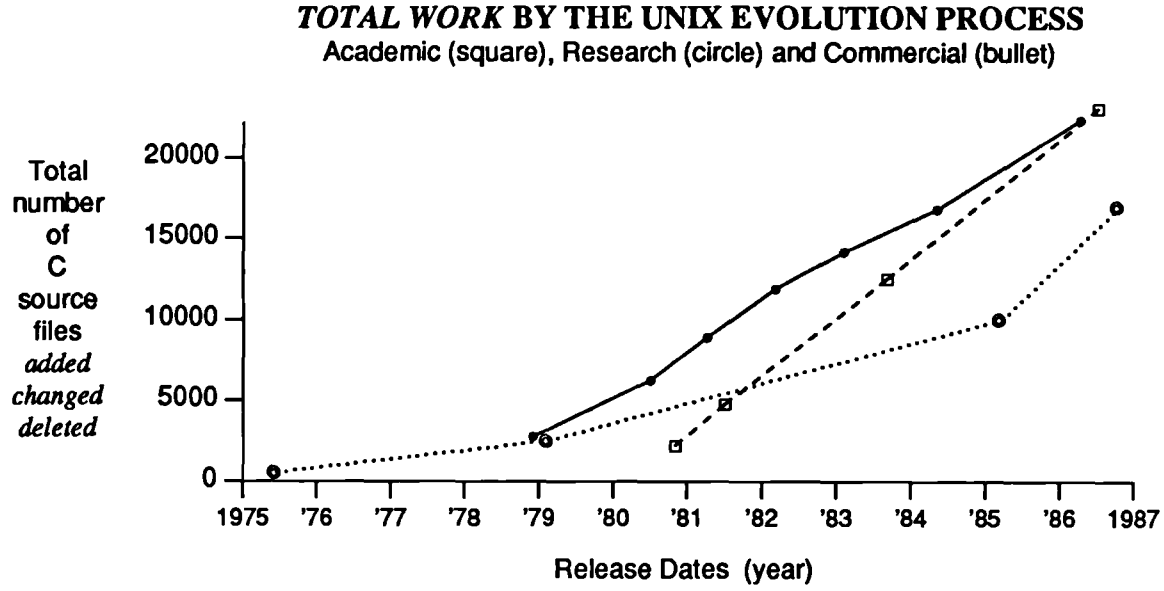
The fit shows a very nice cyclic variation around the regression line, as hypothesised by Lehman. Therefore it can be said, though not with much confidence, that the work-rate of this process has not changed significantly (from 200 files/month) since UNIX/TS 1.0 in spite of numerous organisational and technological changes.



The time plot shows the difference in the average (which only has statistical meaning for supported UNIX) work-rate of the three processes. The graph also brings attention to the fact that, effectively, *instantaneous* work-rate for the process is being measured, at the release point. This measure is more susceptible to fluctuation than a measure of total work done.

A total work measure also removes the inaccuracies in measuring work *for* releases. It has been assumed that the work for a release is carried out in the interval immediately prior to it, however, in practise, it is started several releases prior to the target. This *release overlap* problem disappears when "total work done" is measured.

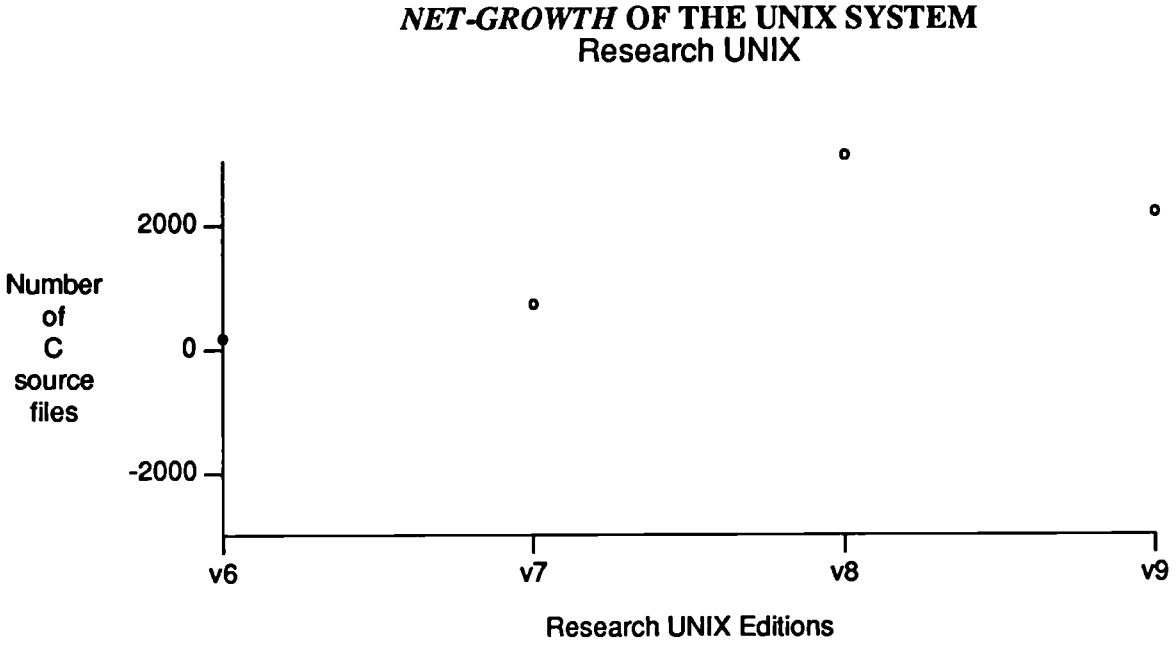
Derived from the same data as the previous graph, it adds the work done for this release (without dividing it by the release interval) to the total work done for previous releases. The unit is total number of files added, changed or deleted.



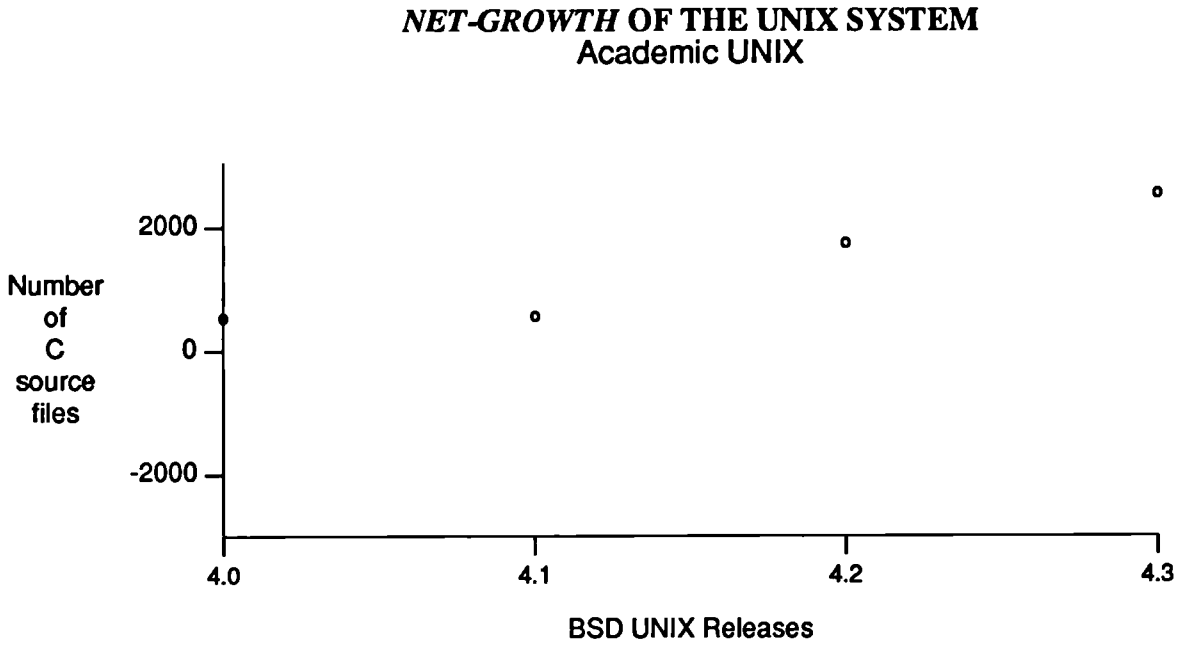
The graph shows a regular trend (positive slope linear) for both supported and academic UNIX. Indeed, the total, at their latest releases, is roughly the same and much higher than research. This is surprising because research has been examined over a much longer time period, but is a result of the low release frequency displayed by the research process.

5.2.5 Growth

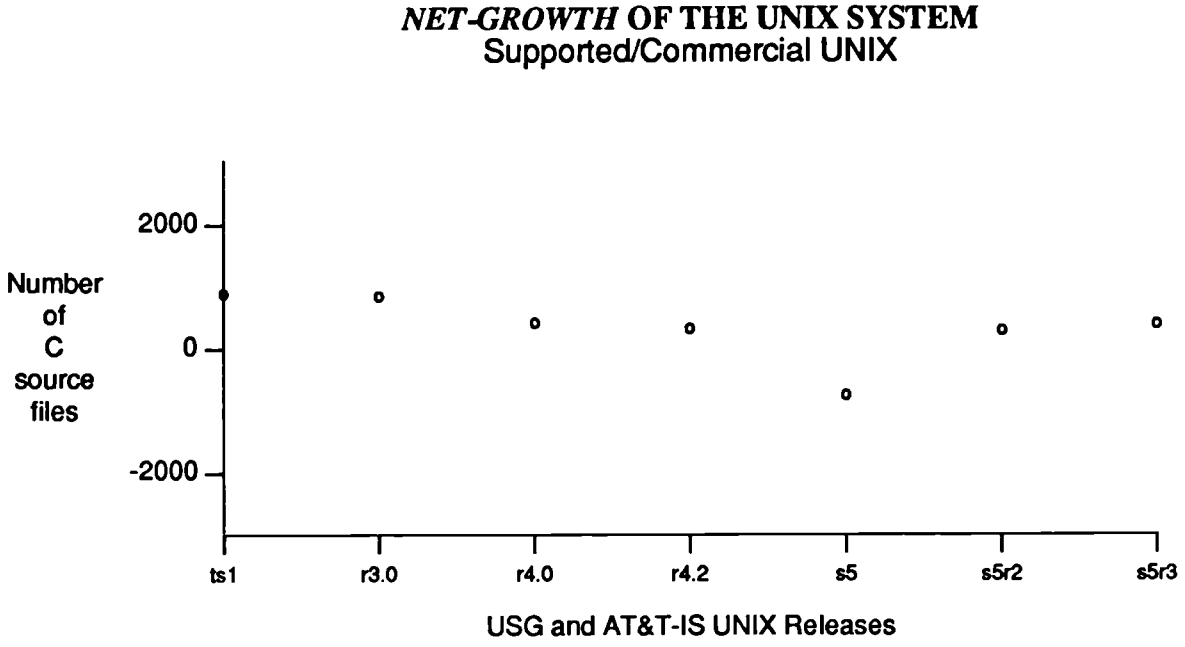
The focus of this subsection is on system growth. Derived from the same data as the 'SIZE' subsection, the plots here show the *difference* in size between successive releases, plotted against the successor release. This metric is measured in number of 'C' source files and positive values imply size increases and negative size decreases. The time taken for the size change is *not* taken into account so the plots are not of growth rate so are not simply plots of the instantaneous slope of the 'SIZE' graphs, rather they are meant to depict inter-release growth.



There is no visible trend here. The low growth between v5 and v6 is expected because there were no major changes in that time frame. The interval between v6 and v7 was used to remove machine specific 'magic numbers' from the code generally clean up the code to make it more portable. Hence, there isn't a large visible size increase at that point. The large peak at v8 has already been explained in the 'SIZE' section. The surprising, since v9 is regarded by the CSRC as a cleanup release, growth between v8 and v9 is 'explained' by the difficulties in obtaining am accurate assessment of v9. Unlike previous releases, there isn't a distinct distribution tape to go with the v9 release (manual- see Chapter 3) and the source measures given throughout this chapter are only a guesstimate of what, from the CSRC system, "would be" on the distribution tape if there was one. The content has, perhaps, been over-estimated.

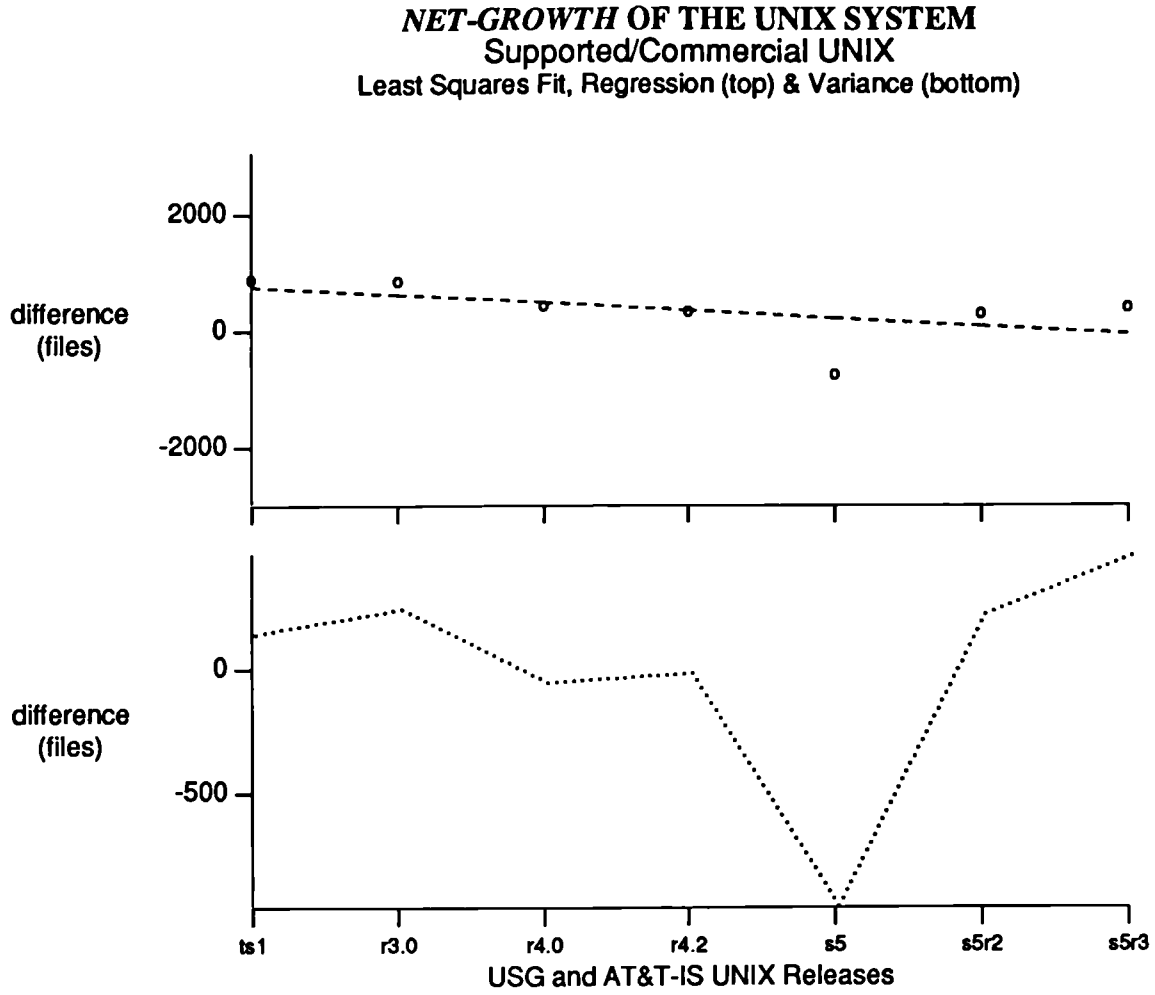


Apart from being monotonically increasing, there isn't a clear trend and the very limited number of data points do not justify further investigation. The graph does, however, show some unexpected behaviour. 4.0 BSD was supposed to be a major feature release (as was 4.2) but the graph does not show a significant increase between 3.0 and 4.0 while it does for the interval between 4.1 and 4.2. The most surprising aspect is the extremely high growth between 4.2 and 4.3 because 4.3 was announced as a cleanup release for which one would not expect a large size increase (like 4.0 -> 4.1). Perhaps the programming team could not (sufficiently) keep their hands off the code in the unexpectedly long release interval (see 'RELEASE INTERVAL' subsection).

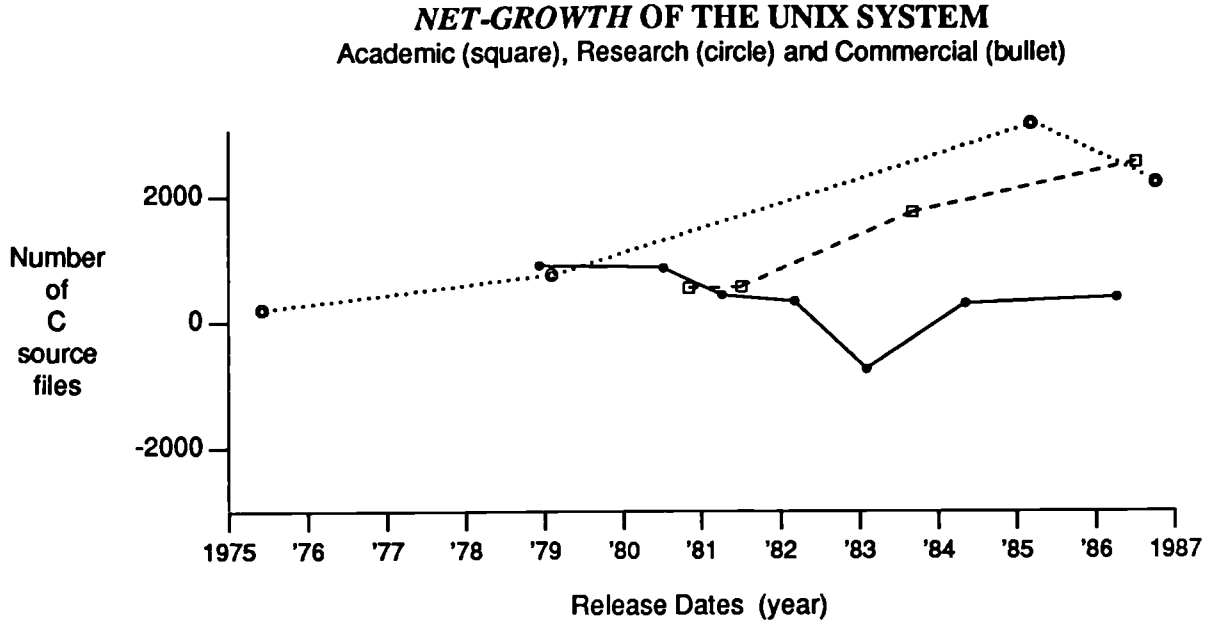


Noticeable is a sharp drop in size at the release of System V. The negative growth in supported/commercial UNIX between r4.2 and s5 is the major point of discontinuity in all plots of this process and warrants an explanation. In preparation for the first commercial release of UNIX (to be offered with full support), the UDL embarked in a major cleanup operation on the code: Bell specific software (e.g. which was not licensed for outside use) was removed from the 'standard' system, machine specific duplicated code was removed[4] and the code tree was generally cleaned up to standard where it would be able to compete against other UNIX systems available commercially. Since then some other unbundling has taken place (see Chapter 3) but scale of the operation for System V is unlikely to be repeated.

4. Prior to that, machine specific source for several supported machines for provided on *all* distribution tapes.



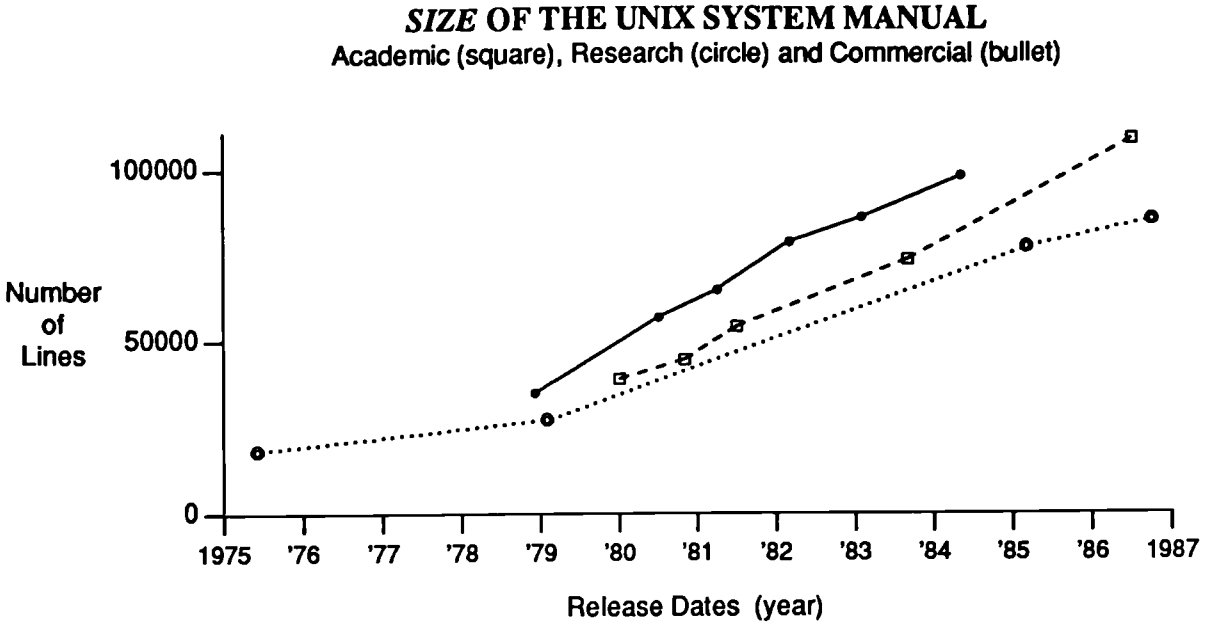
The regression line shows that the least squares fit for the data is a reasonable one, with a very gentle negative slope. This implies that the content of releases has very slightly decreased over the years, indeed if the s5 point is removed (and this can be justified as explained above), the inter-release growth (an indication of release content) is within fairly well defined bounds of 300 -800 files per release.



The compression of scale with the time plot above doesn't give us any more information except making the declining inter-release growth of supported UNIX more visible. The least squares fit for this (not shown here) is similar to the RSN one above.

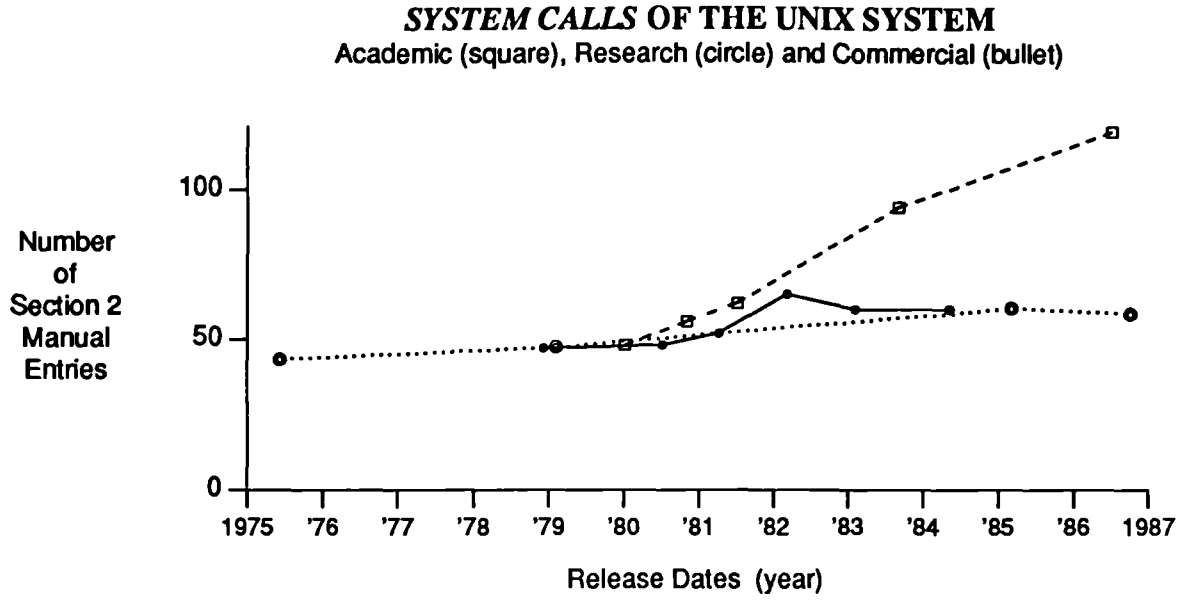
5.2.6 Documentation

A brief analysis (the plots are in Appendix C) of successive editions of the UNIX Programmer's Manual shows that manual in all three branches are steadily growing.



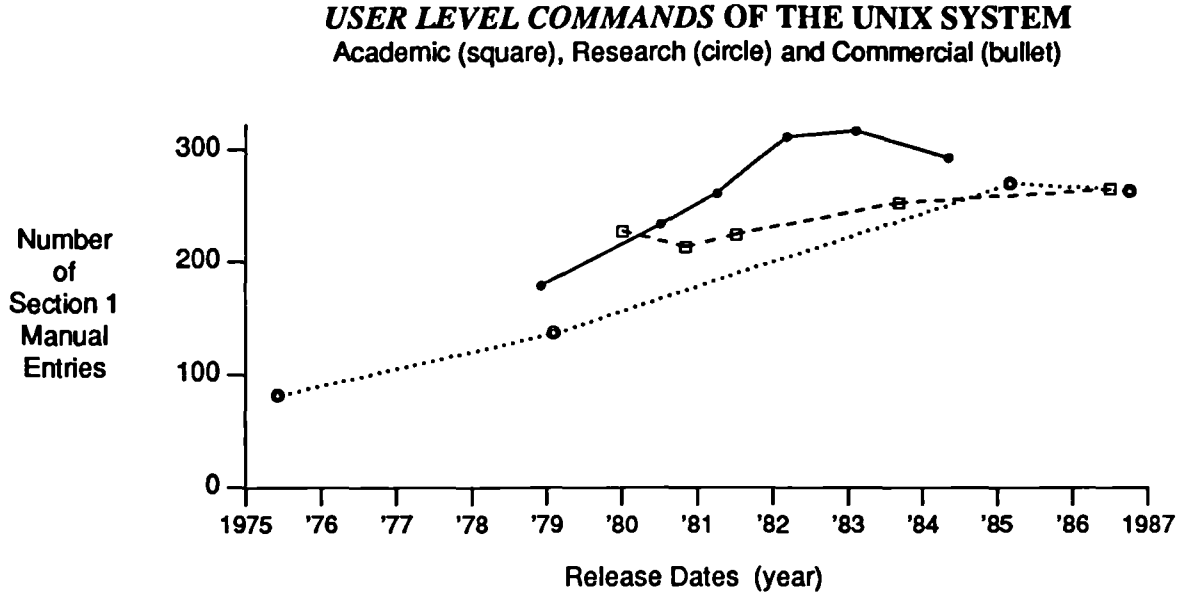
The plot (plotting the number of lines of the manual's **troff** source) shows, visually, linear growth with respect to time, with degrees, in all three. The interesting thing is that the manuals of supported UNIX[5] are consistently larger than BSD manuals which are, in turn, consistently larger than Research manuals. This contrasts with their respective sizes (see Size section in this chapter) and implies that documentation size does *not* reflect code size. This is assuming that documentation keeps pace with coding.

5. The on-line manual for System V Release 3 was not made accessible to this study.



A look at the number of system calls[6] documented in the manual confirms the assertion of several researchers in CSRC that there has been a concerted effort not to needlessly tinker with the kernel. The plot shows almost no rise for both Research and Supported systems. The steady rise in the BSD stream is worrying in that it implies that the kernel is being substantially changed and little effort is directed towards keeping the kernel interface simple.

5. In both, this and the next, graphs the count is the number of genuine entries, i.e. multiple entries are not counted.
6. A system call is an entry point into the system kernel.



This graphs clearly shows the effects of post-commercial unbundling on Commercial UNIX. The plot shows that the number of commands available to user (reflecting the amount of applications software) has reduced since commercialisation. Surprisingly, it still remains higher than the other two, which are about the same. BSD was always thought of (the impression gained at USENIX conferences) as providing a lot of free software.

5.3 PROGNOSIS FOR THE FUTURE OF UNIX

Operating systems have been known to degenerate in time and shown to follow a statistically regular evolution pattern [LEH80][CHO81]. The task of this project in general, and this chapter in particular, is to determine whether UNIX is suffering from the same problems as the systems described in above papers.

This section concentrates on predicting the future evolution pattern of the three UNIX systems examined in the preceding section and offering *numerical* recommendations for their respective programming teams. The validity of the concepts involved is discussed in the next section.

5.3.1 Necessary conditions for the predictions

To be able to successfully extend the models presented in the previous section, and hence predict the systems' future behaviour, the evolution patterns to date must exhibit statistically regular behaviour. Since this report is primarily concerned with understanding the process, the patterns need not withstand rigorous statistical analysis but there must be enough to go on, at the minimum a visible trend.

5.3.2 The Research Stream

With as few as five points for most graphs, we can not even begin to analyse the data, statistically. Especially since (in all cases) there is no visible trend. However, some general observations can be made.

The *size vs time* plot does not show the growth rate (vs time) of the system to be slowing down, indeed, if anything, it seems to be accelerating. This indicates that structural complexity is *not* increasing. This is reinforced by the *module inter-connectivity vs RSN* graph, which shows that successively smaller proportions of the system have had to be modified to introduce new facilities.

Furthermore, the increase in work-rate (at the end of the *work-rate vs RSN* graph) to well over twice the previous highest level suggests that the CSRC programming team (and management) is substantially in control of the project. It remains to be seen whether this peak has resulted in an abnormal increase in system bugs.

5.3.3 The Academic Stream

Like the research stream, there are not enough BSD/UNIX releases to draw statistically meanful conclusions from. However, unlike research, a pattern is visible in two graphs.

The size of BSD/UNIX (see *size vs time* graph) appears to be increasing linearly with time, at the rate of about 850 files per year. Furthermore, this implies that there are no complexity problems. However, the *module inter-connectivity* plot does show a slight rise in the impact to previous releases by new releases.

Surprisingly (since there have been many staffing changes), there seems to be a linear relationship between total work done by the BSD/UNIX process and time. The slope of the *total work vs time* plot works out to be approximately 3,700 files handled per year. For the last two releases, the team has been working at the rate of 300 files handled per month.

5.3.4 The Supported and Commercial Stream

This is the only stream which provides enough plot points to justify some statistical analysis, though the number of points (7-8) still do not make the results statistically significant. Trends are just visible in a number of plots.

The *size vs RSN* plots shows the size of the system to be increasing roughly at 280 files per major release (determined from the slope of the fitted regression line). Since commercialisation this seems to have gone up to 340 files/release but there aren't enough points to say that with

confidence. There is some evidence of increased structural complexity since the rate of size increase with respect to time has come down a bit since commercialization (to 215 files/year from 510 files/year calculated from the slopes of the *size vs time* graph). The *module inter-connectivity* graph also shows an increase in complexity since the release of System V (and interestingly a decrease before that). This is further reinforced by a visible increase in the release interval, as shown in the *release interval vs RSN or time* plots.

The work rate also seems to be fluctuating cyclically (with a cycle of 3-4 releases) around a mean of 200 files handled per month (see *work-rate vs RSN* and the corresponding least squares (LSF) plots). This impression of a constant work-rate is reinforced by the *total work vs time* plot which shows the total work done by the supported/commercial UNIX groups to be increasing at a fairly steady 2,700 files handled per year.

Furthermore, the inter-release growth (an indicator of release content) values of the system seem to have dropped slightly (see *net growth vs RSN* and its corresponding LSF plots). But if the negative value at System V is removed (since the peculiar circumstances of its release are unlikely to be repeated again), a fairly level average inter-release growth rate of about 500 files per release is obtained.

5.3.5 Recommendations

Past experience has shown that if the average inter-release growth rate is exceeded by more than twice its value, severe reliability problems have occurred. Similarly, it has been demonstrated that work-rates exceeding the average value do not allow the programming team enough time sufficiently check their code.

Therefore the following limits are recommended:

Berkeley UNIX	A work-rate limit of 600 files handled per month.
AT&T-IS UNIX	A work-rate limit of 400 files handled per month and an inter-release growth limit of 1,000 files.

Even though the figures above have a weak statistical basis, previous studies [LEH80] [CHO81] have shown that this trend continues regardless of staffing, programming technology and organisational changes.[7] Therefore, the respective managements of the programming teams must

allow for the above.

5.4 EVALUATION OF THE THEORY OF PROGRAM EVOLUTION

This section examines the impact of the UNIX findings on the validity of Lehman's concepts.

Structure of this section

The "Theory of Program Evolution" has already been described in Chapter 2. This section will briefly repeat the characteristics expected of software systems by the 'laws' and then see how accurately these describe the behaviour of the three UNIX systems, taking each 'law' in turn. Since the same results are being discussed (from a different point of view) in this section as the previous one, some repetition is unavoidable.

5.4.1 Characteristics expected by the 'laws'

The first law says that programs grow substantially during their lifetime which implies that a plot of system size versus time or release sequence number should show an increasing trend.

The second law states that programs suffer from increasing structural complexity. This implies that a plot of system size versus time should show (bearing in mind the first law) continuing growth *but at a declining rate*. Furthermore, a plot of the fraction of system changed during a release versus time should also show an increasing trend.

The third law says that software evolves in a statistically regular way which means that all graphs of process and system attributes should have statistically determinable and predictable shapes. Lehman also views the software process as a self-stabilising feedback driven system which would make the system size versus RSN plot show linear growth with a superimposed ripple.

The fourth law states that average work-rate of a software process remains the same throughout its lifetime. So, a plot of the total number of files handled versus time would be expected to show a linear growth trend. A plot of the work-rate achieved in individual releases should a cyclic fluctuation over a constant mean in the sense that an extremely high value should be followed by a low value but the majority should be concentrated around a mean value.

---

7. It must be said, however, that the statistics displayed by the data in this study have turned out to be weaker than in the studies referred to.

Similarly, the fifth law states that the release content will remain constant. This implies that a plot of net incremental growth per release should show similar behaviour to that of work-rate per release.

5.4.2 Behaviour displayed by the UNIX systems

1st law (Growth)

All three UNIX systems appear to be growing strongly. There is only one point, out of a total of 18, at which the size of the system reduces: at the release of System V in the supported/commercial stream.

2nd law (Increasing Complexity)

- There is no evidence of increasing structural complexity in Research UNIX.
- The size of BSD/UNIX is, also, not increasing at a decreasing rate but the impact of a new release on the system is increasing, indicating an increase in complexity.
- The evolution of supported/commercial UNIX seems to have two phases, in this regard. There is a marked difference in growth-rate (vs time) before and after commercialisation but seems not to have dropped after that. Similarly the proportion of system changed by new release plot shows a drop until commercialisation but a sharp rise thereafter. So, there is evidence that complexity has increased in this stream, *after commercialisation*.

3rd law (Self-stabilising Feedback)

Only in the supported/commercial branch of UNIX is there the slightest evidence of self-stabilising feedback: the size of the system seems to be growing roughly linearly with each release, with a super-imposed cycle of three releases.

4th law (Invariant Work-rate)

- The work-rate of the Research UNIX process seems to be *increasing* with each successive release, even more so against time.
- There is some evidence of a stable work-rate in BSD/UNIX as the total work done by the BSD/UNIX process seems to be increasing linearly with time. Furthermore, the work-rate per release seems also to have settle down for the last four (out of five) releases.
- The work-rate per release pattern of supported/commercial UNIX is the only one which remotely resembles a ripple super-imposed on a constant average. This is further reinforced by the total work versus time plot, which shows a linear trend.

5th law (Constant Net Incremental Growth)

- Again, no discernible time dependent behaviour is exhibited by the net growth of Research UNIX systems.
- BSD/UNIX, on the other hand, shows an increasing trend but without a determinable shape.
- Apart from one large decrease in size (at System V), all the net growth points hover around a constant average. There does not seem to be any cyclicity in the graph.

5.4.3 Statistically Smooth Behaviour

This is perhaps the most unexpected, and certainly the most significant, of Lehman's hypotheses: that software evolves with a statistically determinable pattern.
The release frequency of both, Research and BSD, processes has been so low that, clearly, no statistically meaningful conclusions can be drawn. At this rate, to get the same number of data points as the OS/360 study [LEH69], which initiated these concepts, we would have to continue modelling Research UNIX for another 15 years, which is clearly impractical. The problem, in this case, is with the resolution of the models. For systems, with low release frequencies we need to model the systems at points in between releases (say, at each master source integration) to get sufficient data to even test for Lehman's hypotheses. However, in historical studies the chances of getting that data will be very remote, as illustrated by this study.

UNIX releases from USG and its linear successors have been much more frequent and allow us to draw some statistical conclusions, described in the previous section. The results are still not statistically significant (only 7-9 points) so Lehman's hypothesis turns out to be, at best, a weak one.

5.4.4 Conclusions

The dominant observation is the marked difference in behaviour of the three systems. In addition, it seems that Lehman's concepts describe the evolution pattern of commercial software processes best and pure research worst, with supported and academic processes in between.

More investigation is needed to discover if the differences observed in this study are genuine and representative of their cultures (research, academic and commercial respectively). We also need to find out how (or if) the cultural differences affect the scope of the theory. These issues are addressed in the next chapter.

In successfully making some numerical predictions, the previous section demonstrates the usefulness of these concepts. Clearly, this is possible only when the 'laws' apply.

Chapter 6
PROGRAMMING CULTURES

"That men do not learn very much from the lessons of history is the most important of all the lessons that history has to teach."
(ALDOUS HUXLEY, *Collected Essays*, 1959)

6.1 INTRODUCTION

The main aim of this project is to see that, given its history, is it possible to predict the future behaviour of the UNIX system(s). As shown in the previous chapter, statistical projections can only be attempted for the supported/commercial branch of the UNIX evolution tree, in spite of modelling the more well established Research branch for a longer time.

Indeed the marked difference in behaviour of the three streams suggests that environmental and cultural factors play an important role in the dynamics of a particular software process. Research in this area has largely ignored this issue. The task of this chapter is to explore the environmental situations in different programming cultures (specifically research, commercial and academic) and how that affects the dynamics of the respective software process.

6.2 ENVIRONMENTAL FACTORS EFFECTING THE PROGRAMMING PROCESS

This section discusses the characteristics of the programming process, presents the typical (based on observation of Unix houses and descriptions by others) situation present in a development and research environment and speculates on what implications this has for statistical modelling. Each characteristic is discussed in turn.

6.2.1 Measures of Quality

The measures that projects take to keep the staff informed about progress and reward good individual performance are directly related to the overall objectives of the software project.

In a research environment, for instance, where the software is there simply to try out or embody new ideas, the emphasis will be very much on elegance. Productivity will be measured by the number of papers published and the critical response received. Since the code is frequently published in the papers, the researcher is usually interested in coming up with the most elegant solution to the problem and is not shy of throwing away code. Whenever there is a tradeoff to be made between elegance and efficiency, elegance takes precedence [KER84]. Indeed large implementations, which take special cases into account but at the expense of obscuring the main theme, are not encouraged [RIT79]. Of course research programs can take the liberty of not taking every exception into account while the reliability of commercial software is more critical.

The situation is very different in a development environment. A large development project has to keep track of how much progress is being made since contractual deadlines have to be met. Most of the cost or time estimation techniques such as [BOE81], [BAI81], [PUT80] rely directly or indirectly on estimates of product size, measured in some variation of lines of source code. Even a brief glance at a typical programmer's resume will reveal that he regards program size as the definitive performance yardstick.[1] Hence productivity is measured in lines of code produced per unit effort. Complexity of the code is frequently measured in pages of documentation [BOE80], [FEL77], [NEL78][2] and the quality of the product is measured in number of bugs per line of code [MCI88].

Smaller development projects, for instance in computer service departments, are not so concerned about efficiency but rather they aim to keep the service that they supply running and measure their performance by the rate at which they solve reported problems. In other words, they are concerned with effectiveness.

---

1. Phrases like "I was involved several major (>5000 LOC) projects in the last year" or "I produced 30,000 lines of code during my four years at university" are frequently found in resumes posted on the UNIX electronic news network.
2. Other complexity metrics [HAL77], [MCC76] are derived from source code and have proven to be directly proportional to size, these and other better (but not as yet widely used) metrics are discussed in the *Methodologies* Chapter.

Major software projects in academic institutions, like BSD/UNIX [FER83] and CMU/MACH [RAS87], have two main purposes: as examples for teaching undergraduates and as research vehicles for postgraduates. This is, of course, in addition to providing a computing service to the students and staff (in the case of operating systems). When used as an example the code is expected to follow good programming practise and hence the emphasis is on elegance etc. However, the postgraduate students are, by definition, less experienced than professional researchers and are time restricted since they are more concerned about getting their degrees, quality of the implementation is not a high priority with them. If the software is providing a service, it has a further goodness criteria: performance. Students are known to thrash the system [MER85], therefore it is important, in the eyes of the institution, to squeeze as much performance as possible from the limited resources.

The measures will have a profound effect on the dynamics of the process since the programmers will seek to maximize the attribute being measured. Furthermore if these measures are available then the tendency will be for some managers to *rely* on them for assessing the quality of the code (as distinct to the quality of the product) and not look at the code at all (unless the measure is so indistinct {e.g. elegance} that it is necessary to examine the code). This may not necessarily be beneficial to the product; for example: an insightful implementation in 1/4 the number of lines of code with 1/2 the number of errors and 1/10 the documentation would fall short on every development measure above, yet is better in every respect!

6.2.2 Feedback mechanisms

Perhaps the most significant contribution of the Evolution Dynamics studies is the view of the programming process as a self stabilising feedback system [LEH76]. It follows, therefore, that the affects of the environment on the feedback mechanisms, if significant, will greatly affect the programming process.

By definition, a development project has to respond to its customers needs otherwise it will fail. Since customers' needs are constantly changing (see discussion in Chapter 2) the development organization sets up elaborate mechanisms for recording and responding to the needs of its customers. The mechanisms usually take the form of some sort of automated problem reporting system which will typically keep track of all user queries and the "repair" (for no action may be taken) status.[3] The development organization is forced to adopt these formal mechanisms because

it typically has a large user body and the customers, who are paying for the development and maintenance work, may want to query the status of their requests.

Feedback mechanisms are informal for research systems since the research group does not *have* to respond to the needs and wishes of its external (i.e. external to the research centre) users. Nevertheless, research systems are licensed externally (e.g. UNIX) and, even if the licensing fee is minimal, the researchers do judge their success by the critical acclaim they receive to their system. The main point of significance being that the system is shaped according to their own desires not the users. They are not accountable to the users so there is no requirement to record change requests etc. or indeed to take any action on them. The research group can not be totally blind to the user community, of course, because then no one will want their system or accept their papers, rather the "mother knows best" attitude is adopted.

Major computer programs developed in academic institutions are similar to research systems in that one of their functions is to embody the research performed by postgraduate students working towards their degrees. However, since most of the work is financed by grants (e.g from SERC, Alvey, DARPA, NSF) or industrial organisations, some means of recording their wishes are developed. But they are usually less formal than those in commercial environments.

The three situations (or hypotheses) presented above are for projects which result in general purpose software systems (e.g. operating systems) which are then sold or licensed widely. In particular, in the case of sponsored research (e.g. BSD/UNIX and DARPA) the user community is *much* larger than the sponsoring body. It is feedback from the wider users that is informally recorded in research or academic environments. In development situations, *all* paying customers are effectively sponsoring the work.

6.2.3 Release Mechanisms

Since this report is interested in the programming process, as a whole, from outside-in [LEH74], the only external view of the process is the sequence of releases. Indeed, in most studies of this nature, the system is modelled at release points. Therefore it is important to study the effects of the environment on way the software is released to the users.

---

3. All the supported UNIX groups had such systems and they are described in the *Methodologies* chapter. Furthermore the organisations studied in [LEH69], [CHO81] and [HOO75] also had systems for tracking changes, requests and so on.

The release[4] is the primary output of a development group and is the standard means of exporting the code from the programmers to the customers. So, all development work is geared towards producing the release. In the case where there is a maintenance contract the customer will be anxious to get the modifications he requested and the maintenance manager will be under pressure to get the next release (which will contain those changes, for changes typically get packaged into releases) out as soon as possible. Even when there is no maintenance contract, the marketing and sales staff will be keen to get the release out as soon as possible as either the companies revenues will depend on the sales of the next release which, in turn, will depend on whether the release misses the market niche or not. In either case the result will be short intervals between releases and, therefore, high release frequency.

Since research software doesn't have to respond to the needs of its external users, there is little continuous pressure to produce the next release. The typical situation is likely to be that software is rapidly produced by research programmers, consequently rate of internal evolution of the system is high. If the software is not in wide use outside the research centre, then the research centre staff will not be much bothered about getting a coherent release ready for the outside users. Valid users who do want the software simply take a dump of what is on the research centre machine etc. Release numbers follow the manual editions and these come out when, in the opinion of the research gurus, the system has changed sufficiently to warrant a re-publication of the manual[5], which is bound to be not that often. The situation alters slightly if the system has a large user community outside the research centre. Then it becomes organizationally unmanageable to allow all the external users to take dumps of the research centre machines. In this case the research centre staff is forced to produce a coherent release (in the form of e.g. a distribution tape and manual) and give it to someone else for handling licenses etc. To produce this kind of release, the research centre staff has to, temporarily, go into development mode. Since this distracts from their main research activities, it is an infrequent occurrence. Indeed, this only happens, when the researchers want to share major discoveries (or progress) with the outside world. In between these system releases other software may come out of the research centre as independent products (e.g. between v7 and v8 **awk** [AHO78] and **ditroff** were released). The pattern resembles that of any established researcher where we see a number of papers coming out on a particular subject and then we see a book being published (e.g. in the field of program

4. 'Updates' are also counted as releases.

5. The manual reflects the state of the system as is: it is literally a snapshot.

evolution dynamics a number of papers were published between 1969 - 1983, then a book came out in 1985). The papers are published when the researcher when he discovers something new or makes some progress. The books come out when either the field suddenly becomes very popular or important the researcher discovers something very major.[6]

Again academic software is found occupying the middle ground. There is another factor driving the releases here, the sponsoring body. In return for financing the project, the body may want to see progress being made every so often, and requires coherent releases of the software product by those specified dates. But these kind of contracts tend to be generous so the release frequency is bound to be low, say slightly higher than research.

All the release strategies described above assume a large general product (such as an operating system) with a large disparate user community. In the case, where the system is for a very small set of users with relatively static requirements (e.g. the some of the systems described in [HOO75] and [CHO81]), 'releases' would not be expected at all. Instead, the changes would be implemented directly and sent to the (few) customers as soon as they would be completed (i.e. tested etc.).

6.2.4 Change mechanisms and strategies

The previous section discussed how the changes implemented by programmers get to the end users and how the environment affects the frequency with which this is done. The following paragraphs will discuss the effects of the environment on the implementation of changes to the system.

The typical large system will have several customers and the development organization will usually be in the process of enhancing the system. There will be several versions of the system with the bulk of the work being done on the latest one.[7] If a bug is discovered (say, by a customer) in an old version, the project can not simply send out the new version, the bug must be fixed for all supported versions [GLA78]. Hence there is a requirement for development projects

---

6. This cycle is clearly visible in the case of Research UNIX. In the early days (1969 - 1975), since it was new and interesting, the system evolved very rapidly with re-publications of the manuals (hence new 'releases') every six or so months. Then it became very popular (outside the research centre) and release tapes had to be prepared. Between 1975 and 1985 only two releases came out, but each one being a major advance on the previous one. Since then another edition of the manual has come out but without a tape. It has come out after a relatively short time interval and is an evolutionary, rather than a revolutionary step, signifying a return to the early days.

7. This condition is satisfied by all the operating systems studied: IBM OS/360, DOS/360, ICL VME/B and UNIX.

to have (automated) systems which facilitate these version controlling and change managing tasks. While these mechanisms aid traceability, accountability and statistics collection, they slow down the implementation of changes. Lets consider a typical scenario:

> Developer discovers a bug and reports the problem via the problem-reporting (P.R.) system. The system sends the P.R. to the support manager who (possibly after a committee discussion) assesses the problem, decides who should fix it, test it, integrate it, etc. The chosen fixer is notified and so on.

This kind of change control strategy is beneficial when the programmer team is very large (and varying in competence) and the impact of changes (on the rest of the system) must be carefully assessed before they are made and work must be optimally assigned. But it does slow down the process of changing the code and it takes longer to implement decisions. This is particularly infuriating for the competent programmer who, upon discovering a bug, already knows a fix for it but instead of going ahead and fixing it has fill out the various forms and hand it over to the bureaucracy. So, in the presence of these procedures it takes longer to change the system and hence the system changes less often. Also, to keep the internal development environment stable, important when different sub-systems depend on each other, there are internal releases as well as external ones.[8] This further slows down the process of change and encourages staff to conform to the company way of doing things. It also discourages programmers from suggesting changes.

Since the research group, by definition, doesn't support any version of the system there is no requirement to keep old versions accessible or the source under control. Also, the software is written to satisfy the needs and interests of the research team (as opposed to the users) a formal problem-report-driven change control strategy is unnecessary. Free from all these bureaucratic hindrance, the programmers can change the code very rapidly. Since they are writing the code for themselves, there is little difficulty in understanding the requirements. Here the change strategy is very simple, as demonstrated by the typical scenario:

---

8. For example, this is the situation at Imperial Software Technology [IST88], a UNIX Software House in London.

> Person *ken* writes a piece of code, say a new tool, he announces (via electronic news) that there is a new tool, what it does and where it is. Researcher *dmr* uses this tool and finds a bug, he goes over to *ken* and tells him about it. *ken* decides if he has enough time (or interest) to fix it, if so, he goes ahead and makes the change, if not, he asks *dmr* to make it. If *dmr* agrees, the bug is fixed, if not, the bug is advertised (via enews).

It is as simple as that, bugs and suggestions are reported to whoever is responsible for a particular bit of code (its well known - within the research centre - who owns what) and he makes or commissions the change. Changes take a short time to implement and since there are no internal releases as such, the changes are exposed to the research centre users very quickly. This type of environment fosters the rapid prototyping kind of approach (rather than structured top-down) to programming and works well only when most of the members in the team are competent. The resultant code is also very susceptible to personal style and experimentation. The down side is that this results in an unstable environment: there is no guarantee that the system tomorrow will in the same state as today or even compatible. Furthermore, very few or no records are kept. This makes it very difficult to trace something back and hindsight statistics collection becomes a major challenge.

The students who contribute software to the academic system typically don't keep their source under control since they are concerned with the primary research. The integration team, which is responsible for building the system from various contributions from different research groups and students and then distributing the system on the campus machines, may want to keep track of the goings on (this is the case with BSD/UNIX and CMU/MACH) just to find their way through the large amounts of software being contributed.

In a service-type environment, where the software is developed for a very restricted set of users who (usually) report to the same management as the developers, the number of change requests do not justify elaborate change control strategies, however, [HOO75] and [CHO81] have indicated that such organisations have on-line systems for tracking design and code changes. All versions of the source is also kept on-line in such situations.

6.2.5 The Product

The previous section discussed how the environment affects the way changes are implemented to the source code of a system. This section will investigate how the programming environment influences what goes into the system and how this affects the dynamics of the process.

In a service type environment, the users dictate what changes are made to the system. Since the users and the developers share the same management, approving the changes is not a problem. [CHO81] indicates that technical considerations rarely outweigh the functional ones, after all, the software is there purely to serve the specified users. Since the two groups (users and programmers) are likely to share the same hardware, user machine changes are likely to change the software.

In the case of operating systems (which can absorb completely independent pieces of code as utilities), a research centre system reflects *all* the research activity going on in the centre [CRO78] [MCI86]. Theoreticians interested in algorithms and data structures[9] affect the performance of the software while those interested in a particular area of computer science write sub-systems or utilities[10] which may be released separately but eventually become part of the general operating system [MCI87]. Research systems are very sensitive to hardware and software configurations in the research centre but completely insensitive to the configurations of the external users. Most of the what enters the system is the result of some research and is technically approved by the programmers, there are no marketing or commercial considerations to deride from the technical merit. The tendency is for the code to be changed if and only if there is a substantial orthogonal advance to be made by change. Hence research operating systems will accumulate many utilities and new developments but will shed support for old devices. The code for each specific function will however, be quite succinct (compared to code achieving similar functionality in other environments).

The primary motivation for producing software in commercial environments is money (which equates to power for development groups serving large corporations - like in the early days of USG/UNIX). This is linked to the sales of the software product and related contracts. Hence sales and marketing departments have a say in what goes into the software system. This effects what new developments end up in 'the' software system. If, in the opinion of the marketing team, there is more money to be made in selling a new (sometimes even an existing - see unbundling) utility separately, it is not included in 'the' system.[11] Also, to maximise sales, they have to provide what

9. For example, the file comparison algorithm in **diff** or the pattern matching algorithm in **egrep**. Also the work reported in [AHO74] and [AHO75] influenced tools like **yacc** and **lex** and eventually led to the development of the portable 'C' compiler.

10. For example, the device independent typesetter driver **ditroff** or the pattern scanning and processing language **awk**.

11. Motor manufacturers have the same problem: the engineers may feel the car must have power steering as standard, but the marketing team feels that the resultant price rise will be critical and offer it as an option, the result is that the standard car is slated in the press for having a heavy steering! AT&T have starting selling utilities separately which were once in the system, e.g. the text processing software.

the customer wants, even if it is at odds with the technical teams wishes.[12] Similarly, in the case of supported operating systems, development groups are forced to support hardware and software configurations which they themselves have given up and may not be interested in. Another marketing phenomenon which hinders technical development is marketing by feature lists. Marketing teams are much more comfortable with features that they can easily list and quantify. This tends to breed development of products which have several specific features rather than a few general ones which may make the existence of the specific features irrelevant. So, one would expect development systems to have larger manuals, but less 'standard' utilities.

6.2.6 Staffing, organization and management

This section will explore the effects of some facets of staffing on the dynamics of the programming process.

* Competence and experience
* Turn-over
* Size and structure

Size and Structure

The larger the size of the programming team, the more complex the organization structure, the longer the communication channels, the higher the likelihood of misunderstandings and the longer the bureaucratic delays in getting things done. Furthermore the sheer size of the system will make process less responsive to individual decisions. It is more likely to have an inertia of its own. More people working on the same piece of code, compounding configuration management problems. Adding to the confusion is the problem of partitioning the work-load which necessitates defining interfaces. In a large organization, the management may not, in certain circumstances, fully understand all the technical issues. For instance, when evaluating a tool, a non-technical manager will judge a it purely by its user interface, and will not look at the code. He may of course rely on other 'measures' of code quality, see first section in this chapter. This may not necessarily be a problem since he can easily ask for advice but it is an example of an

12. An example from the motor industry will clarify this: recently American car buyers have tended to prefer front-wheel drive cars, the large US manufacturers have 'had' to respond by changing their large cars to front wheel drive. Technically, rear wheel drive is more appropriate for large, powerful cars. In software: say an operating system doesn't support real-time processing very well and to change the system to do so will spoil its structure, but the marketing bosses insist on it because it will boost sales.

additional overhead which is less likely to occur in small organisations or groups. Another point worth mentioning is that studies have shown that computer programmers are task-oriented [SOM82], i.e. they are motivated primarily by the work itself, hence they prefer to be left alone to do their own thing and hence they perform better in an environment which fosters that i.e. one that has a very laid back management.

Management and Team Competence

Clearly the competence of the programming team will have a direct significant impact on the quality of the end product (and the process). It also effects the sensitivity of the performance of the team to its management. If most members of the group are experienced (and competent) then they can function properly with minimal management direction, provided the organization structure does not inhibit direct and informal communication between the members. In such a situation the younger members of the team also rapidly learn the trade. The idea is that each one shapes his own product for its own good, peer pressure will ensure its 'quality' and this will lead to betterment of the whole system. Management becomes critical to the success of the team in a situation where the majority of the programming team is not experienced. Particularly important becomes the ability of the management to use the experience of their senior staff to lift the performance of the whole team. Some typical techniques used are Chief Programmer Teams, Structured Code Walkthroughs, Programming Standards and Quality Assurance Programmes. All these techniques are geared towards providing more structure, inhibiting individual freedom for the sake of ensuring a minimum standard for the whole system. The individual programmers are more likely to be in control of the process and the system in the first situation described, while the structure and mechanisms in the second will work towards reducing variance and hence make it more likely to exhibit regular evolution trends.

Staff Turnover

Another factor which could effect the dynamics of the programming process is turnover of the programming staff. If staff changes are frequent, the programmers will be required to modify someone else's code, they are unlikely to fully understand the code in the limited time that they have available[13], and this is likely to slow down the change process. In fact, programmers are encouraged (even required) to write documentation for their code precisely to aid these

13. Indeed, it is likely that by the time they become adequately (what ever that may be) familiar with the code, they will move on to other projects, groups or companies.

transitions, hence further stabilizing the change environment. Another direct outcome of this frequent hand over of code is a 'familiarity limit' in that programmers only get so familiar with the code. Furthermore, since they will be changing projects so often, it is unlikely that they will grow attached to any particular piece of code ("this is my baby") and therefore will not take as much care of it as they might.

The reverse phenomena apply when staff turnover is low. The programmers kind increasingly familiar with the code hence equivalent changes take less time to implement, the programmers grow a sense of loyalty to their products and hence take care of it well, they are less bogged down with having to teach new staff their code (because there is no new staff taking over their code). This also has a disadvantage in that the programmer may get bored with having responsibility for the same code for a long period.

Therefore, it seems, that a process with a high staff turnover is much more likely to display regular evolution patterns than a process with low staff turnover, in which case the programmers have more control over the product.

UNIX situations

Based on observations of the UNIX houses (see history chapter) and other software centres it appears that research groups have very low staff turnover, medium size (~70 people), very laid back management and very competent[14] staff. Commercial software groups, on the other hand, have a surprisingly high turnover rate, large team size (~200), active management and a mixture of a few competent programmers amongst many comparatively not-so-competent staff. Academic projects, by definition, have very high staff turnover, as most of the work is done by students, who are, obviously, inexperienced (having a maximum of 3-5 years programming experience) though possibly competent; management is more active than in research but less than in development.

6.3 CONCLUSIONS

After examining the situations that exist in the different types of programming environments (particularly in the UNIX groups under study), this section will investigate how these combine to make statistical modelling of the programming process possible in some environments but not in

14. For example they have to posses a track record of proven research potential, by, for instance, having a doctorate.

others.

First and foremost there will simply be much more data available in a development environment, so it will be much easier to test for statistical significance. The problem report system, the change management system, performance measurements, timesheets and other management records will all contribute towards ensuring that complete histories of the source, staffing, testing and motivation for changes are available. The most that a research process can be expected to deliver are the release tapes. There will simply be no (or very little) information on the source for the inter-release period. While there will be some information on staffing there will be no information on and how effort is being expended on a particular system (or part of-). There will be no statistics on change requests etc., the only records of motivations will be those published in papers and books etc. There will be more records available in academic environments but possibly not as much as in commercial environments.

The commercial, marketing and management pressures will make system releases from a development group more frequent than from academic or research groups. This higher release frequency will provide more data points to test various hypotheses for development projects of equivalent age as research projects. Furthermore, this makes it more likely that in successive development releases relatively less of the code will be changed than in successive releases of research or academic products, where the chances are that more of the system will be changed than left intact!

A direct corollary of the low release frequency of academic and research projects will be relatively large inter-release intervals. This will allow the respective programming teams more time to clean up and re-structure the code etc. and plenty of time to prototype the code. Hence, such projects are less likely to suffer from increasing structural complexity of their products. Also, unless information is available for the inter-release period (and this is not likely to be the case for research projects - see above), a dynamic picture of the processes will not be available (or the picture will be less dynamic than in development environments) and, because of that, it is questionable whether such processes can be usefully modelled in this way.

The relatively high staff turn-over in development and academic environments, coupled with the way work is typically assigned in the presence of commercial pressure, will work towards establishing a limit to how familiar the programmer gets with the code. There will be no such "conservation of familiarity" in research groups.

The various bureaucratic mechanisms to record problem reports and control source changes, in development environments, serve to delay the implementation of decisions. Furthermore the large team size and correspondingly complex organization structure (including union -like bodies) act

as process stablizers. Indeed the whole setup is geared towards stability, particularly, if the organization has a large investment in the product. All this makes the process less responsive to *individual* management decisions. The lack of formal mechanism for change management, in research environments, and the small and simple staff structure makes it possible to implement decisions rapidly and makes the process much more under the control of individual managers and programmers. The situation in academic environments lies in between the two mentioned above.

In summary, the hypothesis is that there won't be enough data to statistically test for trends and variances of a programming process in a research or academic type environment. Furthermore even if there was enough data to test; they would be less likely to suffer from deteriorating structure or follow the model predicted (of statistically regular trends and variances) by "The Theory of Program Evolution".

It appears, therefore, that the property of being *self-regulating feedback driven* describes a system which is large, complex and expensive; where the momentum of the project (caused by people, budget, practices) and the general smoothing properties of the organization determine the rate of progress and the fate of the project [LEH74]. As shown above, not all non-trivial software is constructed in such environments.

Chapter 7
UNDER PRESSURE

"Either the human being must suffer and struggle
as the price of a more searching vision,
or his gaze must be shallow and without intellectual revelation."
(THOMAS DE QUINCY, "Vision of Life", *Suspira de Profundis*, 1845)

7.1 INTRODUCTION

The UNIX system has been put under tremendous pressure by its own success in the operating system market. It is under pressure structurally, as requests from its rapidly widening user base cause the system to be changed rapidly. It is also under pressure commercially, as various new standards movements compete to influence the future direction of UNIX.

Another field under pressure is that of software metric modelling and in particular Lehman's "Theory of Program Evolution". In spite of recent activity, the field is struggling to establish its credibility [FEN88], [BAC88]. Furthermore, there are still question marks on the validity of Lehman's concepts [LAW82], which researchers are still waiting to have removed [CON86], [BOE84].

This chapter summarizes the discoveries and conclusions of this thesis by

- validating the concepts of different process types (Chapter 6) on the previous systems studied
- pooling the UNIX findings (Chapter 5) with previous critical evaluations of the "Theory of Program Evolution" to propose an updated theory

In addition, a brief analysis of the current market forces on UNIX is presented to supplement the structural evaluation of UNIX given in Chapter 5. In doing so it intends to relieve some of the

pressure mentioned above.

7.2 CULTURAL ASPECTS OF THE PROCESS

After examining cultural differences in various aspects of the programming process (like release strategy, configuration management and staffing), Chapter 6 concluded that software developed in a commercial environment is more likely to display the dynamics predicted by Lehman, than software developed in a purely research environment. This is because a commercial software process is likely to have a higher release frequency; higher staff and work turnover; and complicated configuration management.

7.2.1 Test against known behaviour

The task of this section is to briefly test the hypothesis presented above against the known behaviour of systems examined in previous ED studies.

Product environment ratings

Of the software systems studied previously, the ones with the most 'commercial' environment were the operating systems developed by IBM and ICL. Clearly, these are major proprietary operating systems whose commercial success was closely linked to the companys' success in selling hardware, their main revenue earner. All these systems were in many thousands of installations (it is estimated [CAR88] that 370, OS/360's successor, currently captures 21% of the market). Meeting the needs of their many and varied users was obviously vital to the continued commercial success of the systems and the development organisations.

The other products are less well known. However, Omega was a commercial transaction system in use, earning revenue for its developer. Similarly CCSS was installed at a few sites, however its usage environment was less commercial since it was used purely by the military. The other two systems (Executive and BD) are also in very limited use by customers with very similar requirements. In both cases the development of the systems is driven to a large extent by the users.

Process environment ratings

The information available on most of the development projects associated with the products above is sketchy at best (ICL's VME/B is the exception). It is known, however, that the teams used by the hardware manufacturers for the operating systems were large, well organised and necessarily led by profit conscious management with structured configuration management mechanisms to record users' wishes and control the sources etc. The other teams were probably less structured since they had to deal with fewer and more similar users.

Composite commerciality ratings

Hence all the above systems are commercial in nature. In decreasing order of "commerciality" they would be rated:

OS/360, DOS, VME/B, OMEGA, CCSS, EXECUTIVE, BD

Observed behaviour

[LAW82] and [KIT82] confirm that the vendor operating systems do appear to display the characteristics predicted by Lehman, to a greater extent than the others.

7.2.2 Recommendations for further work

Clearly the analysis above is rather superficial and imprecise. This is largely due to subjective evaluation of the products and processes, which is necessarily imprecise.

Therefore, what we need are well defined, quantitative measures of this property *commerciality*.

- The product environment measures would take into account aspects such as number of users, type of usage and the product's revenue earning importance to the development organization.
- The process environment measures would take into account the team size and competence, staff turnover, work assignment strategies and communication paths, release strategy and configuration management procedures etc.

Research should be focused at how such a metric could be constructed (and what it should be called). As a starting point we suggest a weighted sum of the above attributes in which the higher the end result, the higher the probability that the system will exhibit regular trends and smooth behaviour etc.

7.3 PROPOSALS FOR THEORY OF PROGRAM EVOLUTION

Since the formulation of the five laws [BEL76], summarizing the "Theory of Program Evolution", other researchers have critically examined the concepts proposed [CHO81], [LAW82] and [KIT82] (see Chapter 2 for details).

This section pools their conclusions with those of this study (presented in Chapter 5) to see if any of the laws can be easily changed to bring them more in tune with reality.

7.3.1 Continuing Growth

All the studies support the first law, of continuing change. Indeed all observe growth during the useful life of the system. UNIX systems appear to behave similarly so the first law can, perhaps, be more strongly worded:

> A program that is used and that as an implementation of its specification reflects some other reality, undergoes continual change and *becomes progressively larger*. This process continues until it becomes more cost effective to replace the program.

7.3.2 Increasing complexity?

All agree that complexity is very difficult to measure. However, using existing (including Lehman's) measures [KIT82] and [LAW82] conclude that only some (the minority) of the systems exhibit increasing complexity. Only commercial UNIX shows clear signs of increasing complexity, so the second 'law' needs further investigation to see why some, but not others, show increasing complexity. There is no clear trend in the ones which do show increasing complexity (OS/360, Omega, and the commercial releases of UNIX) have no identifiable distinguishing feature.

The problem may lie in the measures of complexity used. Both Lawrence and Kitchenham have commented that Lehman's measure (Fraction of modules handled) is very susceptible to large release intervals or depends on the release content being roughly constant. A new metric proposed in [BEN79] showed no increase in value for the Executive system. Perhaps it is safest to stick to Lehman's original *declining growth rate* measure. Metrics proposed by other researchers (see Chapter 4 for review) are not much better.

In any case, a lot of research is being carried out to come up with measures of program complexity [INC88] and it is worthwhile waiting for a quantum leap in that field to re-test the 2nd law before rejecting it. It will be even more worthwhile to identify a set of distinguishing characteristics to separate systems which seem to show increasing complexity from those which don't. However that is likely to require more data than is available.

7.3.3 Smooth behaviour in commercial systems

[KIT82] observes some signs of invariant work rate but [LAW82] concludes that there is no statistical evidence for statistically smooth behaviour. [CHO81] prefers to say that the trends show no time dependent Some parts of the UNIX evolution process do seem to be changing smoothly, however, the basis for statistical inference is weak. Since, 'laws' 3-5 do not withstand rigorous statistical scrutiny, it can be concluded that *all* software evolution is *not* statistically modelable.

An analysis of different factors affecting the software evolution process (in Chapter 6) indicated that a *commercial* software process, i.e. where the product is a revenue earner for its development organization, is more likely to exhibit the phenomena described by Lehman than software developed in a less structured environment such as research. Therefore, the following change is suggested to the "Fundamental Law of Program Evolution":

> Program Evolution, *in the presence of commercial pressures*, is subject to a dynamics which makes the programming process, and hence measures of global project and system attributes, self-regulating with statistically determinable trends and variances.

It remains to be seen if even commercial software evolves statistically smoothly. The paramount problem with testing for smoothness etc. will be lack of data, if software continues to be modelled only on release basis. Constructing higher resolution models will become easier as time goes on because more and more industrial projects are keeping more and more data (as it becomes relatively cheaper).

7.3.4 Recommendations for further work

- As mentioned above, research is needed to come up with better measures of software complexity and then use those to determine what characterises the systems which show increasing complexity.
- Also, more data needs to be collected on more systems with differing environments to see if indeed commercial software development more likely to have an inertia of its own. DEC's VMS is the obvious contender for a commercial system while CMU's MACH or MIT's X/Windows could possibly represent research projects.
- A project following the recommendations in [BAS80] would have enough data to enable automatic construction of continuous models. Such models could then be increasingly calibrated as the product and process age.

7.4 ARE METRIC MODELLING TECHNIQUES USEFUL?

To a large extent the usefulness of the concepts discussed in this thesis depend on their validity. If their validity is accepted, then the implications for project planning and control are obvious (described in Chapter 5).

However, as explained above, there is still a long way to go in terms of process and product measures and data collection before the kind of metric models constructed here can be relied upon.

At the moment, they merely serve as rough guidelines, at best.

7.5 THE UNIX MARKET PLACE: Survival of the fittest?

One of the conclusions of Chapter 5, after examining the evolution of the system, was that UNIX is reasonably structurally sound at the moment. Of the three branches examined, only the commercial stream is displaying some problems but not nearly to the same extent as those observed in other systems such as IBM's OS/360 [LEH74].

There is little doubt that UNIX will be made to address its few technical weaknesses and this examination has shown that it will be able to respond. Indeed the next major release of UNIX from AT&T (System V release 4) is scheduled to support parallel processing. It will also have real-time capabilities for transaction processing and will be a secure and fault tolerant system [COM88a].

However, technical supremacy is never a guarantee of success in the commercial market, of any product. Witness the success of the top ten selling cars in the UK, none of whom are anywhere near the best in their respective classes [GRE88] and the run-away success of the IBM-PC. Factors such as marketing and perceived customer support play a more important role.

This section briefly describes recent developments in the UNIX market place and puts those into perspective by very briefly reviewing the system's progress since its birth (Chapter 3 describes the history in detail).

7.5.1 The past

In the late sixties, some staff at Bell Labs became annoyed at difficulty encountered while writing computer programs so they began to search for a more friendly programming environment than they had. Not finding one, they decided to write their own: UNIX was born. Shortly afterwards, the system was rewritten in 'C', a high-level language that they themselves had written. At about this time the popularity of DEC's new PDP-11 increased rapidly throughout Bell Labs (because it was cheap, it allowed the individual group to move away from the centralised mainframes etc.). Impressed by the in-house UNIX, those who obtained PDP-11s, invariably chose UNIX as their operating system.

The quality of its implementation, visible through the on-line source code, made the system easy to understand. Soon several groups started tailoring the system to meet their own needs and building applications on top of it. Since the Research Center could not be relied upon to provide support a UNIX Support Group was set up. But a number of groups continued to work on their own version of UNIX (e.g. PWB, CB/OSG), some of whom built up enough credibility to

compete with USG for potential UNIX 'sales' (since no money was involved, the 'customers' being within Bell Labs). Research UNIX did not enter into the equation since they could not (did not) provide support, which was essential to a number of users (especially those building products on top). By the mid to late seventies the UNIX situation was very confusing in the Labs, there were a number of UNIX variants and with a number of incompatibilities.

To rationalise the situation, high level inter departmental committee, decided to standardise UNIX by having only one supported version.[1] One group would be responsible for it (incorporating the other groups) and it would have the most desirable features from all the UNIX variants. Release 3.0 bore the fruits of this effort which reached completion with System V when all other UNIX efforts within Bell Labs ceased. Other than the Research Center, which continued to go its own way but this did not affect the standardisation scheme since that was only concerned with supported efforts.

This became the commercial (offered with full support) release of UNIX from Bell Labs and is the only major UNIX variant commercially available from AT&T. It is also the dominant UNIX version within AT&T, since Research UNIX is only used in a handful of research departments.[2]

7.5.2 The present

UNIX also became popular outside the Bell System. Since the Research system was offered without support, outside organisations, especially universities, had to and did learn UNIX, and successfully tailored the system for their own needs. When UNIX became the first operating system to be ported, its popularity grew tremendously, since it allowed a vendor independent operating system. As a result of which, several other companies ported UNIX to their own or other people's hardware.

Currently, almost every major computer manufacturer has a version of UNIX for their machines. In addition, a number of software houses have developed their own version of UNIX. Before 1983, the outside versions were mostly derived from Research UNIX or UC Berkeley's BSD/UNIX,[3] since then they are mostly based on System V.

---

1. In addition there was one supported *real-time* version.
2. Although other development efforts have ceased, there are still some installations which use old UNIX variants and have not upgraded to System V.
3. This has become the dominant version for the VAX, since it was the first to take advantage of the machines special features, as described earlier.

This has resulted in a compatibility problem, since a large application developed on one variant will, in general, not run on another, without change. In an effort to help standardise the application-to-operating system interface, and hence reduce customers' porting costs, AT&T (which owns UNIX) brought out the *SV-ID* for the System V Interface Definition [SVI86]. This document specifies the complete operating system environment.

In the same vein, an IEEE Working Group was set up to investigate a Portable Operating System for Computer Environment standard. They took over an earlier effort by the UNIX User Group and is, therefore, based on UNIX. The working group intends to gain wider acceptance for *POSIX*, as the standard is called, from ANSI and ISO. The document [POS87] will address system interface; shell and tools; testing; and real-time issues.

In 1984, major European Computer manufacturers realised that systems in the low end of the market were sold on the basis of applications software rather than hardware. Proprietary operating systems made it difficult to third parties to build applications software. In an effort to aid these third parties (hence penetrate this market), the manufacturers set up *X/Open*. The group's objectives were to define a complete environment for portable applications and intends to deal with issues relating to data management, distributed systems and use of high level languages [X/O87]. It is basically an extension of the SV-ID.

To reduce the number of UNIX variant incompatibilities, AT&T announced in late 1987, that it intended to merge System V with 4.3 BSD and SunOS. In early 1988, AT&T bought a 20% stake in the microsystems' manufacturer Sun and set up a joint group to steer their operating system development. In February 1988, AT&T announced that System V Release 4 will conform to POSIX, SV-ID and X/Open standard. It will also have features from MicroSoft's Xenix, BSD and will therefore provide a single converged UNIX to the market, planned for autumn 1989.

Other computer manufacturer's, worried that Sun was being unfairly favoured by AT&T, set up *Open Software Foundation* (OSF). Led by IBM and DEC, the foundation aims to provide an open software environment. This will include application interfaces, advanced system extensions and a new operating system (reportedly based on IBM's AIX) which will comply with POSIX etc.

So, at the moment, it would appear that the commercial UNIX world is polarised into two camps: AT&T and its supporters; and members of OSF.

7.6 CLOSING REMARKS

It is interesting to note that what happened within AT&T about a decade age is now taking place on a global scale.

The sequence is: popularity of UNIX due to technical merit, lots of variants, efforts to rationalise the situation, research continue to provide the technical state of the art but not suitable for commercial applications.

The difference is that the early variants were managerially within AT&T's control so could be forced to standardise or disband and the current variants are independent. The current situation is made worse by the fact that the groups owning the variants are financially tied in to the commercial success of their systems while the early ones were not commercial (so less cut-throat).

However, the largest software customer in the world (the US Department of Defence) may "persuade" the variants to merge or standardise and thus play the same role as AT&T's upper management in dealing with its internal variants. In fact, the D.o.D. has already stated that all its future applications software must comply with POSIX.

Another analogy worth drawing is with the evolution of the PC market where initially everyone jumped onto the bandwagon but the competition was so fierce that several companies went bankrupt or pulled out of the PC market. The IBM PC, despite its technical inferiority, became the *de* facto standard. Apple, though, managed to survive but everyone else mostly started to build IBM-PC clones.

The UNIX market seems to be going through the same cycle, evidenced by the fierce competition now enveloping UNIX houses, some of whom are now starting to get out, or are being forced out. POSIX is poised to take over from AT&T System V as *the* standard and the market is likely to end up saturated with software that meets POSIX.

Therefore, the concluding lesson of this investigation is a confirmation of the dictum that:

"History repeats itself."

Appendix A
GLOSSARY

This glossary covers major terms and acronyms used in this thesis. The definitions describe usage as in this thesis.

AT&T	The *American Telephone and Telegraph* Company. This term is mostly used (in this thesis) interchangeably with the Bell System.
ATT-IS	AT&T Information Systems, the linear successors of USG and currently responsible for Commercial releases of UNIX, System V.
Academic UNIX	BSD/UNIX releases.
BSD, BSD/UNIX	for *Berkeley Software Distribution*. Software release from UCB were called BSD. UNIX releases from UCB are known as BSD/UNIX.
Bell System	The AT&T empire of companies, including Bell Labs. See AT&T.
CB-OSG	the *Operations Systems Group* at Bell Labs. in Columbus, Ohio (*CB* is the Bell Labs. identification for their Columbus facility). Responsible for CB/UNIX.
CMU	the *Carnegie Mellon University* in Pittsburgh, Penn.
CSRC	the *Computing Science Research Center* at AT&T Bell Laboratories in Murray Hill, New Jersey. Responsible for Research versions of UNIX. This is where UNIX was invented. Also known as Center1127.
CSRG	the *Computer Systems Research Group* in the Computer Science Division of the Electrical Engineering and Computer Science Department in the University of California at Berkeley. Responsible for BSD/UNIX.
Center1127	see CSRC.
Commercial UNIX	System V UNIX releases from UDL and ATT-IS.
ED, PED	The field of Program *Evolution Dynamics*. An area of research to investigate the dynamics of a programming process.
LOC	*Lines of code*, a size measure.
LSF	short form for *Least Squares Fit*.
PWB	*Programmer's Workbench*, the UNIX effort originally based in Business Information Systems Area of Bell Labs., Piscataway, New Jersey.

RSN	*Release Sequence Number*, a pseudo time measure used as the X-axis (the attribute measure is the Y-axis) in most ED studies' plots.
Research Center	see CSRC.
Research UNIX	UNIX releases from CSRC.
SCCS	*Source Code Control System*, a system for keeping records of changes to text files, particularly program source. Developed by the PWB effort.
Supported UNIX	Development UNIX releases, mostly from USG and UDL.
UCB	the *University of California at Berkeley*.
UDL	the *UNIX Development Laboratory* took over the Bell Labs. UNIX *development* effort, from USG.
USG	the *UNIX Support Group* formed in 1973 to provide support to Bell System users of UNIX. Evolved into the UDL.
WECo	*Western Electric Company*, was the licensing arm of AT&T.
s5, s5r...	Short form for System V releases, see Commercial UNIX.
v1..v9	see Research UNIX.

Appendix B
REFERENCES

[AHO74] A. V. Aho and S. C. Johnson, *LR Parsing*, Comp. Surveys, Vol. 6, No. 2, pp. 99-124, June 1974.

[AHO75] A. V. Aho, S. C. Johnson and J. D. Ullman, *Deterministic Parsing of Ambiguous Grammars*, Comm. Assoc. Comp. Mach., Vol. 18, No. 8, pp. 441-452, August 1975.

[AHO78] A. V. Aho, P. J. Weinberger and B. W. Kernighan, *AWK - a pattern scanning and processing language*, Software-Practice and Experience, July 1978.

[AKI71] F. Akiyama, *An example of software system debugging*, Information Processing 71, pp. 353-379, 1971.

[ALB79] A. J. Albrecht, *Measuring Application Development Productivity*, Proc. of the Joint SHARE/GUIDE/IBM Application Dev. Sym., Guide Int. Corp., Chicago, 1979.

[BAC88] R. Bache, N. E. Fenton, R. Tinker and R. W. Whitty, *Software Quality Assurance: A Rigorous Engineering Practice*, Proc. 2nd BCS/IEE Software Eng. Conf., pp. 3-7, July 1988.

[BAI81] J. J. Bailey and V. R. Basili, *A meta-model for software development resource expenditures*, Proc. 5th Int. Conf. Software Eng., pp. 107-116, March 1981.

[BAK72] F. T. Baker, *Chief programmer team management of production programming*, IBM Systems J., Vol. 11, No. 1, pp. 56-73, 1972.

[BAK80] A. L. Baker and S. H. Zweben, *A comparison of measures of control flow complexity*, IEEE Trans. Software Eng., Vol. SE-6, No. 6, pp. 506-512, Nov. 1980.

[BAN88] R. D. Banker and C. F. Kemerer, *Scale Economies in New Software Development*, CISR WP No. 167, Center for Information Systems Research, Sloan School, M.I.T., Feb. 1988.

[BAS79] V. R. Basili and R. W. Reiter, *Evaluating automatable measures of software development*, IEEE NY Poly. Workshop on Quantitative Software Models for Reliability, Compexity and Cost, pp. 117-116, Kianasha Lake, NY, Oct. 1979.

[BAS79a] V. R. Basili and R. W. Rieter, *An investigation of human factors in software development*, IEEE Comp., Vol. 12, No. 12, pp. 21-38, Dec. 1979.

[BAS80] V. R. Basili, *Data Collection*, Tutorial on Models and Metrics for Software Management and Engineering, IEEE Computer Society, Sept. 1980.

[BEL71] L. A. Belady and M. M. Lehman, *Programming System Dynamics or the Metadynamics of Systems in Maintenance and Growth*, IBM Res. Rep. RC 3546, Sept. 1971.

[BEL72] L. A. Belady and M. M. Lehman, *An Introduction to Growth Dynamics*, From Statistical Computer Performance Evaluation, pp. 503-511, Academic Press, 1972.

[BEL74] D. E. Bell and J. E. Sullivan, *Further investigation into the complexity of software*, MITRE Technical Report 2874-2, June 1974.

[BEL75] L. A. Belady and H. Beilner, *Speculations on Program Complexity*, Unpublished Paper, IBM Research, 1975.Reported in [BEL85]

[BEL76] L. A. Belady and M. M. Lehman, *A Model of Large Program Development*, IBM Sys. J., Vol. 15, No. 3, pp. 225-252, 1976.

[BEL79] L. A. Belady, *On Software Complexity*, IEEE NY Poly. Workshop on Quantitative Software Models for Reliability, Complexity and Cost, pp. 90-94, Kianasha Lake, NY, Oct. 1979.

[BEL85] L. A. Belady and M. M. Lehman, *Program Evolution - Processes of Software Change*, Academic Press, 1985.

[BEN79] G. Benyon-Tinker, *Complexity measures in an evolving large system*, Proc. Workshop on Quantitative Software Models for Reliability, Lake Kiamecha, Oct. 1979.

[BOE80] B. W. Boehm, *Developing Small-Scale Application Software Products: Some Experimental Results*, Proc. IFIP 8th World Comp. Cong., pp. 321-326, Oct. 1980.

[BOE81] B. W. Boehm, *Software Engineering Economics*, Prentice-Hall, Englewood Cliffs, New Jersey, 1981.

[BOE84] B. W. Boehm, *Software Engineering Economics*, IEEE Trans. Software Eng., Vol. SE-10, No. 1, pp. 4-21, Jan. 1984.

[BOU78] S. R. Bourne, *UNIX Time-Sharing System: The UNIX Shell*, Bell Sys. Tech. J., Vol. 57, No. 6, pp. 1971-1990, 1978.

[BRA76] R. B. Brandt, *UNIX Program Description*, PD-1c301-01, PG-1c300 issue 2, Bell Laboratories, Jan. 1976.

[CAR62] E. J. Carbato, M. M. Dagget and R. C. Daley, *An Experimental Time Sharing System*, Proc. Spring Joint Comp. Conf. American Fed. Information Processing Soc., pp. 335-344, 1962.

[CAR65] E. J. Carbato and V. A. Vyssotsky, et. al., *A New Remote Accessed Man-Machine System*, Proc. Fall Joint Comp. Conf. American Fed. Information Processing Soc., pp. 185-247, 1965.

[CAR88] R. H. Carlyle, *Open Systems: What Price Freedom?*, Datamation, June 1, 1988.

[CHE75] L. L. Cherry and B. W. Kernighan, *A System for Typesetting Mathematics*, Comm. Assoc. Comp. Mach., Vol. 18, pp. 151-157, March 1975.

[CHE78] E. T. Chen, *Program complexity and programmer productivity*, IEEE Trans. Software Eng., Vol. SE-3, pp. 187-194, 1978.

[CHO77] C. K. S. ChongHokYuen, *Evolution Dynamics and its Application to Two Large Programs*, B Sc Project Report, Department of Computing and Control, Imperial College, London SW7 2BZ, June 1977.Reported in [CHO80].

[CHO80] C. K. S. ChongHokYuen, *A Phenomenology of Program Maintenance and Evolution*, PhD Thesis, Department of Computing, Imperial College, London SW7 2BZ, Nov. 1980.

[CHO83] C. K. S. ChongHokYuen, *A Statistical Rationale for Evolution Dynamics Concepts*, Concordia University, Montreal, Canada, 1983.

[CLA78] C. P. Clare, *A Case Study in Operating System Evolution Dynamics*, M Sc Thesis, Department of Computing and Control, Imperial College, London SW7 2BZ, Sept. 1978.

[CON86] S. D. Conte, D. E. Dunsmore and V. Y. Shen, *Software Engineering Metrics and Models*, Benjamin/Cummings, Menlo Park, CA, 1986.

[COU83] N. S. Coulter, *Software Science and Cognitive psychology*, IEEE Trans. Software Eng., Vol. SE-9, No. 2, pp. 166-171, March 1983.

[CRO78] T. H. Crowley, *UNIX Time-Sharing System: Preface*, Bell Sys. Tech. J., Vol. 57, No. 6, pp. 1897-1898, 1978.

[CUR79] B. Curtis, *In search of software complexity*, IEEE NY Poly. Workshop on Quantitative Software Models for Reliability, Compexity and Cost, pp. 95-106, Kianasha Lake, NY, Oct. 1979.

[CUR79a] B. Curtis, S. B. Sheppard and P. Milliman, *Third time charm: Stronger prediction of programmer performance by software complexity metrics*, Proc. 4th Intl. Conf. Software Eng., pp. 356-360, 1979.

[DAG88] G. Daglish, *Evolution of VME/B: a curve fitting exercise*, Software Reliability and Metrics Newsletter, No. 7, pp. 18-23, Centre for Software Reliability, City Univ., London EC1 0HB, July 1988.

[DAR84] I. Darwin and G. Collyer, *The Evolution of UNIX - 1974 to the Present*, Microsystems, Vol. 57, No. 11, Nov. 1984.

[DAS88] S. K. Das, *UNIX Around the World*, Proc. Spring 1988 EUUG Conf., pp. 1-6, April 11-15, 1988.

[DIJ68] E. W. Dijkstra, *Goto statements considered harmful*, Comm. Assoc. Comp. Mach., Vol. 11, No. 3, pp. 147-148, March 1968.

[DOL76] T. A. Dolotta and J. R. Mashey, *An Introduction to the Programmer's Workbench*, Proc. 2nd Int. Conf. on Software Engineering, Oct. 13-15, 1976.

[DRU82] L. E. Druffel, *Strategy for a DoD Software Initiative*, CSS DUSD(RAT), Washington, DC, 1982.

[DUN79] H. E. Dunsmore and J. D. Gannon, *Data referencing: an emperical investigation*, IEEE Computer, Vol. 12, No. 12, pp. 50-59, Dec. 1979.

[DUN80] H. E. Dunsmore and J. D. Gannon, *Analysis of the effects of programming factors on programming effort*, J. of Sys. and Software, Vol. 1, No. 2, pp. 141-153, 1980.

[ELS78] J. L. Elshoff, *A review of software measurement studies at General Motors Research Laboratories*, Proc. 2nd Life Cycle Management Workshop, pp. 166-171, IEEE, New York, 1978.

[EMD71] M. H. Van Emden, *An Analysis of Complexity*, Mathematische Centrum, Amsterdam, 1971.

[FEL77] C. P. Felix and C. E. Walston, *A method of programming measurement and estimation*, IBM Syst. J., Vol. 16, No. 1, pp. 54-73, 1977.

[FEL78] S. I. Feldman and P. J. Weinberger, *A Portable Fortran 77 Compiler*, Bell Laboratories, Murray Hill, New Jersey, Aug. 1978.

[FEL79] S. I. Feldman, *MAKE - a program for maintaining computer programs*, Software - Practice and Experience, April 1979.First published as BTL CSTR No 57, 1977.

[FEL84] S. I. Feldman, *An Architecture History of the UNIX System*, Proc. USENIX Conf., Salt Lake City, Summer 1984.

[FEN88] N. Fenton, *Software Measurement*, Software Reliability and Metrics Newsletter, No. 7, pp. 35-62, Centre for Software Reliability, City Univ., London EC1 0HB, July 1988.

[FER83] D. Ferrari, *The Evolution of Berkeley UNIX*, Report No. UCB/CSD 83/155, Comp. Sci. Div. (EECS), UC, Berkeley, CA, Dec. 1983.

[FRA79] A. G. Fraser, *Datakit - A Modular Network for Synchronous and Asynchronous Traffic*, Proc. Intl. Conf. on Commun., Boston, MA, June 1979.

[FUN76] Y. Funami and M. H. Halstead, *A software physics analysis of Akiyama's debugging data*, Proc. MRI 24th Intl. Sym.: Software Eng., pp. 133-138, Polytechnic Press, New York, 1976.

[GLA78] A. L. Glasser, *The Evolution of a Source Code Control System*, Proc. Software Quality Assurance Workshop, Nov. 15-17, 1978.

[GRE88] G. Green, *Top 10 Sellers on Trial*, Car, Aug. 1988.

[HAL72] M. H. Halstead, *Natural laws controlling algorithm structure*, SIGPLAN Notices, Vol. 7, No. 2, pp. 19-26, 1972.

[HAL77] M. H. Halstead, *Elements of Software Science*, Elsevier North-Holland, New York, 1977.

[HAN78] W. J. Hansen, *Measurement of program complexity by the pair (cyclomatic number, operator count)*, ACM SIGPLAN Notices, Vol. 13, No. 3, pp. 29-33, March 1978.

[HOG78] T. Hogan, *A Case Study in Evolution Dynamics*, B Sc Project Report, Department of Computing and Control, Imperial College, London SW7 2BZ, June 1978.Reported in [CHO80].

[HOL82] B. D. Holbrook and W. S. Brown, *A History of Computing Research at Bell Laboratories (1937-1975)*, Comp. Sci. Tech. Rep. No. 99, Bell Telephone Laboratories, Incorporated, 1982.

[HOO75] D. H. Hooton, *A Case Study in Evolution Dynamics*, MSc Thesis, Department of Computing and Control, Imperial College, London SW7 2BZ, Sept. 1975.

[HUN76] J. W. Hunt and M. D. McIlroy, *An Algorithm for Differential File Comparison*, Comp. Sci. Tech. Rep. No. 41, Bell Laboratories, Murray Hill, New Jersey, June 1976.

[INC88] D. Ince, *Anti-rust treatments*, Systems International, pp. 21-22, Aug. 1988.

[ITA82] M. Itakura and A. Takayanagi, *A model for estimating program size and its evaluation*, Proc. 6th Int. Software Eng. Conf., pp. 104-109, Sept. 1982.

[IVI77] E. L. Ivie, *The Programmer's Workbench - A Machine for Software Development*, Comm. Assoc. Comp. Mach., Vol. 20, No. 10, pp. 746-753, Oct. 1977.

[JEN83] R. W. Jensen, *An improved macrolevel software development resource estimation model*, Proc. 5th ISPA Conf., pp. 88-92, April 1983.

[JEN86] J. O. Jenkins and A. J. C. Cowderoy, *State of the art survey for software cost-estimation*, Esprit P938 report W/P5 (issue 3), Imperial College Management School, London SW7 2PG, August 1986.

[JEN88] J. O. Jenkins and A. J. C. Cowderoy, *Cost-estimation by Analogy as a Good Management Practice*, Proc. 2nd BCS/IEE Software Eng. Conf., pp. 80-84, July 1988.

[JOH75] S. C. Johnson, *Yacc — Yet Another Compiler-Compiler*, Comp. Sci. Tech. Rep. No. 32, Bell Laboratories, Murray Hill, New Jersey, July 1975.

[JOH78] S. C. Johnson and D. M. Ritchie, *UNIX Time-Sharing System: Portability of C Programs and the UNIX System*, Bell Sys. Tech. J., Vol. 57, No. 6, pp. 2021-2048, 1978.

[JOH78a] S. C. Johnson, *A Portable Compiler: Theory and Practice*, Proc. 5th ACM Symp. on Principles of Programming Languages, pp. 97-104, January 1978.

[KER78] B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Prentice-Hall, Englewood Cliffs, New Jersey, 1978.

[KER84] B. W. Kernighan and R. Pike, *The UNIX Programming Environment*, Prentice-Hall, Englewood Cliffs, NJ, 1984.

[KIT81] B. A. Kitchenham, *Measures of programming complexity*, ICL Tech. J., pp. 298-316, May 1981.

[KIT82] B. A. Kitchenham, *Systems evolution dynamics of VME/B*, ICL Tech. J., pp. 43-57, May 1982.

[KNU76] D. B. Knudsen, A. Barofsky and L. R. Satz, *A Modification Request Control System*, Proc. 2nd Intl. Conf. Software Eng., Oct. 13-15, 1976.

[KUH82] W. W. Kuhn, *A Software lifecycle case study using the PRICE model*, Proc. IEEE NAECON, May 1982.

[LAW82] M. J. Lawrence, *An examination of evolution dynamics*, Proc. 6th Int. Conf. Software Eng., pp. 188-196, Sept. 1982.

[LEH69] M. M. Lehman, *The Programming Process*, IBM Res. Rep. RC 2722, December 1969.

[LEH74] M. M. Lehman, *Programs, Cities and Students - Limits to Growth?*, Imperial College Inaugural Lecture Series, Vol. 9, pp. 211-229, May 14, 1974.

[LEH76] M. M. Lehman and F. N. Parr, *Program Evolution and Its Impact on Software Engineering*, Proc. 2nd Intl. Software Eng. Conf., pp. 350-357, 1976.

[LEH77] M. M. Lehman and J. Patterson, *Preliminary CCSS System Analysis Using Techniques of Evolution Dynamics*, Working Papers Software Life Cycle Management Workshop, pp. 324-332, Airlie, VA, 1977.

[LEH78] M. M. Lehman, *Laws of Program Evolution - Rules and Tools of Programming Management*, Proc. Infotech State of the Art Conf. on "Why Software Projects Fail?", pp. 11/1-11/25, April 1978.

[LEH80] M. M. Lehman, *Programs, Life Cycles and Laws of Software Evolution*, Proc. IEEE Spec. Iss. on Software Eng., pp. 1060-1076, Sept. 1980.

[LEH82] M. M. Lehman, *Program Evolution*, Proc. Sym. on Empirical Foundation of Computing and Information Sciences, Georgia Inst. Tech., Nov. 3-7, 1982.

[LEH84] M. M. Lehman, N. V. Stenning and W. M. Turski, *Another Look at Software Design Methodology*, Software Engineering Notes, Vol. 9, No. 2, pp. 38-53, April 1984.

[LEO88] J. Leonard, J. Pardon and S. Wade, *Software Maintenance - Cinderella is still not getting to the Ball*, Proc. 2nd BCS/IEE Software Eng. Conf., pp. 104-106, July 1988.

[LES75] M. E. Lesk, *Lex — A Lexical Analyzer Generator*, Comp. Sci. Tech. Rep. No. 39, Bell Laboratories, Murray Hill, New Jersey, October 1975.

[LIE80] B. Lientz and E. B. Swanson, *Software Maintenance Management*, Addison-Wesley, 1980.

[LIM75] A. L. Lim, *Preface to ED Analysis*, IBM, April 1975.Reported in [CHO80]

[LON78] T. B. London and J. F. Riser, *A UNIX Operating System for the DEC VAX-11/780 Computer*, Bell Labs. Tech. Memorandum 78-1353-4, July 1978.

[LYC78] H. Lycklama and D. L. Bayer, *UNIX Time-Sharing System: The MERT Operating System*, Bell Sys. Tech. J., Vol. 57, No. 6, pp. 2049-2086, 1978.

[MAR84] R. L. Martin, *The UNIX System: Preface*, AT&T Bell Lab. Tech. J., Vol. 63, No. 8, pp. 1571-1572, 1984.

[MCC76] T. J. McCabe, *A complexity measure*, IEEE Trans. Software Eng., Vol. SE-2, No. 4, pp. 308-320, Dec. 1976.

[MCI86] M. D. McIlroy, *The UNIX Success Story*, UNIX Review, Oct. 1986.

[MCI87] M. D. McIlroy, *A Research UNIX Reader: Annotated Excerpts from the Programmer's Manual, 1971-1986*, Comp. Sci. Tech. Rep. No. 139, Bell Laboratories, Murray Hill, New Jersey, June 1987.

[MCI88] M. D. McIlroy, *Private Comm.*, Feb. 1988.

[MCK85] M. K. McKusick, *A Berkeley Odyssey - Ten years of BSD history*, UNIX Review, Jan. 1985.

[MCM78] L. E. McMahon, *SED - A Non-interactive Text Editor*, Bell Laboratories, Murray Hill, New Jersey, August 15, 1978.

[MER85] D. Merrit, R. Toxen and K. Arnold, *Fear and Loathing on the UNIX Trail '76*, UNIX Review, Jan. 1985.

[MOR73] S. P. Morgan, *Minicomputers in Bell Laboratories Research*, Bell Laboratories Record, Vol. 51, pp. 194-201, July/August 1973.

[MOT77] R. W. Motley and W. D. Brooks, *Statistical prediction of programming errors*, RADC-TR-77-175, Rome Air Development Center, Griffiss AFB, NY, May 1977.

[MUS75] J. D. Musa, *A Theory of Software Reliability and its Application*, IEEE Trans. Software Eng., Vol. SE-1, No. 3, Sept. 1975.

[MUS80] J. D. Musa, *Software reliability measurement*, J. of Sys. and Software, pp. 223-241, 1980.

[MYE77] G. J. Myers, *An extention to the cyclomatic measure of program complexity*, SIGPLAN Notices, Vol. 12, No. 10, pp. 61-64, 1977.

[MYE78] G. J. Myers, *Composite/Structured Design*, Van Nostrand Reinhold, New York, 1978.

[NEL66] E. A. Nelson, *Management handbook for the estimation of computer programming costs*, AD-A648750, Syst. Develop. Corp., Oct. 31, 1966.Reported in [BOE84].

[NEL78] R. Nelson, *Software Data Collection and Analysis ar RADC*, Rome Air Development Center, Rome, NY, 1978.

[ORG72] E. I. Organick, *The MULTICS System*, M.I.T. Press, Cambridge, Mass., 1972.

[OSS77] J. F. Ossanna, *NROFF/TROFF User's Manual*, Comp. Sci. Tech. Rep. No. 54, Bell Laboratories, Murray Hill, New Jersey, April 1977.

[OTT79] L. M. Ottenstein, *Quantitative estimates of debugging requirements*, IEEE Trans. Software Eng., Vol. SE-5, No. 5, pp. 504-514, Sept. 1979.

[OVI80] E. I. Oviedo, *Control flow, data flow, and program complexity*, Proc. IEEE Comp. Software and Applications Conf., pp. 146-152, Nov. 1980.

[POS87] *POSIX P1003.2 Draft 4*, The IEEE, New York, NY, Nov. 16, 1987.

[PAR72] D. L. Parnass, *On the Criteria to be used in Decomposing Systems into Modules*, Comm. Assoc. Comp. Mach., Vol. 15, No. 12, pp. 1053-1058, Dec. 1972.

[PIK84] R. Pike, *The Blit: A Multiplexed Graphics Terminal*, AT&T Bell Lab. Tech. J., Vol. 63, No. 8, pp. 1607-1632, 1984.

[POT82] D. Potier, J. L. Albin, R. Ferreol and A. Bilodeau, *Experiments with computer software complexity and reliability*, Proc. 6th Intl. Conf. Software Eng., pp. 94-103, 1982.

[PRA88] R. E. Prather, *Comparison and Extention of Theories of Zipf and Halstead*, The Comp. J., Vol. 31, No. 3, pp. 248-252, 1988.

[PUT78] L. H. Putnam, *A general empirical solution to the macro software sizing and estimating problem*, IEEE Trans. Software Eng., pp. 345-361, July 1978.

[PUT80] L. H. Putnam, *SLIM System Description*, Quantitative Software Management, Inc., McLean, VA, 1980.

[PUT84] L. H. Putnam, D. T. Putnam and L. P. Thayer, *A tools for planning software projects*, J. of Sys. and Software, Vol. 5, pp. 147-154, Jan. 1984.

[RAS87] R. E. Rashid, A. Tevanian, D. B. Golub, D. L. Black, E. Cooper and M. W. Young, *Mach Threads and the UNIX Kernel: The Battle for Control*, Summer Conference Proc., USENIX Association, 1987.

[RIO77] J. S. Riordan, *An Evolution Dynamics Model of Software Systems Development*, Software Phenomology - Working Papers of the Software Life Cycle Management Workshop, Airlie, Virginia, Aug. 1977.

[RIT74] D. M. Ritchie and K. Thompson, *The UNIX Time-Sharing System*, Comm. Assoc. Comp. Mach., Vol. 17, No. 7, pp. 365-375, July 1974.

[RIT78] D. M. Ritchie, *UNIX Time-Sharing System: A Retrospective*, Bell Sys. Tech. J., Vol. 57, No. 6, pp. 1947-1969, 1978.Also in *Proc. Hawaii International Conference on Systems Science*, Honolulu, Hawaii, Jan. 1977.

[RIT79] D. M. Ritchie, *The Evolution of the UNIX Time-sharing System*, Lecture Notes on Computer Science, Vol. Language Design and Programming Meth., No. 79, Springer-Verlag, 1979.

[RIT84] D. M. Ritchie, *A Stream Input-Output System*, AT&T Bell Lab. Tech. J., Vol. 63, No. 8, pp. 1897-1910, 1984.

[RIT84a] D. M. Ritchie, *Reflections on Software Research*, Comm. Assoc. Comp. Mach., Vol. 27, No. 8, pp. 758-760, Aug. 1984.

[ROC75] M. J. Rochkind, *The Source Code Control System*, IEEE Trans. on Software Eng., Vol. SE-1, No. 4, pp. 364-370, Dec. 1975.

[SVI86] *System V Interface Definition (SVID) Issue 2*, AT&T Customer Information Center, Indianapolis, IN, 1986.

[SCH79] N. F. Scheindewind and H. Hoffman, *An experiment in error detection and analysis*, IEEE Trans. Software Eng., Vol. SE-5, No. 3, pp. 276-286, May 1979.

[SHE83] V. Y. Shen, S. D. Conte and D. E. Dunsmore, , IEEE Trans. Software Eng., Vol. SE-9, No. 2, pp. 155-165, March 1983.

[SHE88] M. Sheppard, *A critique of cyclomatic complexity as a software metric*, Software Eng. J., pp. 30-36, March 1988.

[SHO83] M. L. Shooman, *Software Engineering*, McGraw-Hill, New York, 1983.

[SIM69] H. A. Simon, *The Science of the Artificial*, MIT Press, 1969.

[SMI80] C. P. Smith, *A Software Science analysis of programming size*, Proc. ACM National Comp. Conf., pp. 179-185, Oct. 1980.

[SNY74] A. Snyder, *A Portable Compiler for the Language C*, M Sc Thesis, M.I.T., Cambridge, Mass., 1974.

[SOM82] I. Sommerville, *Software Engineering*, Addison-Wesley, 1982.

[STE88] I. Stewartson, *UNIX System V.3 and Beyond*, Proc. Spring 1988 EUUG Conf., pp. 161-178, April 11-15, 1988.

[STR67] J. M. Stroud, *The fine structure of psychological time*, Annuals of New York Academy of Science, Vol. 138, No. 2, pp. 623-631, 1967.

[THO75] K. Thompson, *The UNIX Command Language*, pp. 375-384, Infotech International Ltd., Nicholson House, Maidenhead, Berkshire, England, March 1975.

[VER87] J. M. Verner and G. Tate, *A Model for Software Sizing*, J. of Sys. and Software, Vol. 7, pp. 172-177, 1987.

[WEI70] G. M. Weinberg, *Natural Selection as Applied to Computers and Programs*, General Systems, Vol. 15, pp. 145-150, 1970.

[WIL72] R. I. Wilson, *Introduction to Graph Theory*, Academic Press, New York, 1972.

[WOL74] R. W. Wolverton, *The cost of developing large-scale software*, IEEE Trans. Comput., pp. 615-636, June 1974.

[WOO79] M. R. Woodward, M. A. Hennell and D. Hedley, *A measure of control flow complexity in program text*, IEEE Trans. Software Eng., Vol. SE-5, No. 1, pp. 45-50, Jan. 1979.

[WOO80] C. M. Woodside, *A Mathematical Model for the Evolution of Software*, J. Sys. and Software, Vol. 1, No. 4, pp. 337-345, Oct. 1980.

[WOO81] S. N. Woodfield, V. Y. Shen and H. E. Dunsmore, *A study of several metrics for programming effort*, J. of Sys. and Software 2, 2, pp. 97-103, Dec. 1981.

[X/O87] *X/Open Portability Guide Issue 2*, Elsevier Science Publishers, Amsterdam, The Netherlands, Jan. 1987.

[YAT88] *Supporting the UNIX System*, The Yates Perspective, Vol. 1, No. 3, pp. 1, 6-8, Nov./Dec. 1982.

[ZOL81] J. C. Zolnowski and D. B. Simmons, *Taking the measure of program complexity*, Proc. of the National Comp. Conf., pp. 329-336, 1981.

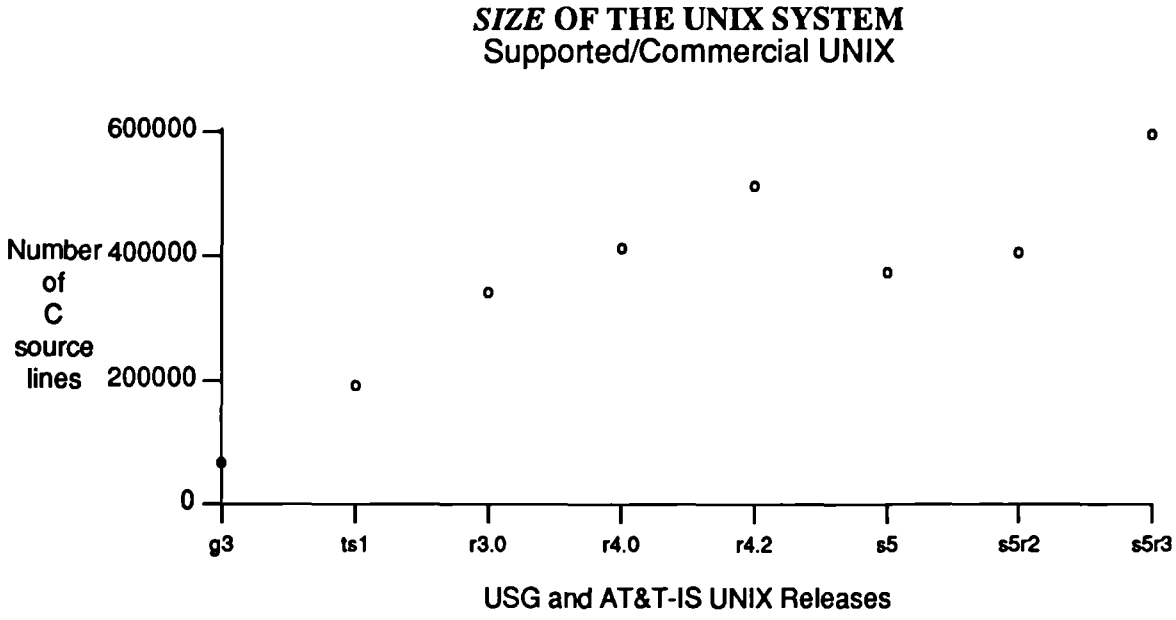
Appendix C
MISCELLANEOUS PLOTS

C.1 INTRODUCTION

This appendix contains the plots that were *not* used to model the UNIX evolution process. The plots are given below merely to illustrate their appearance, and show the evolution of only the supported UNIX stream.

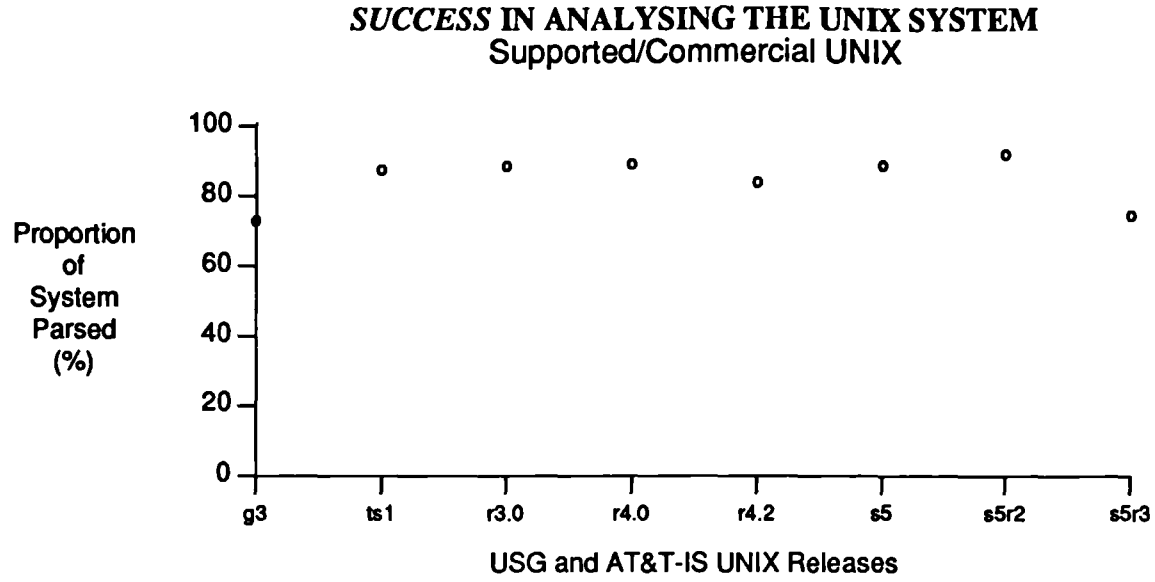
C.2 SIZE

The *number of lines* plot gives no further information, than the *number of files* plots shown in Chapter 5, other than a slight exageration of vertical scale.

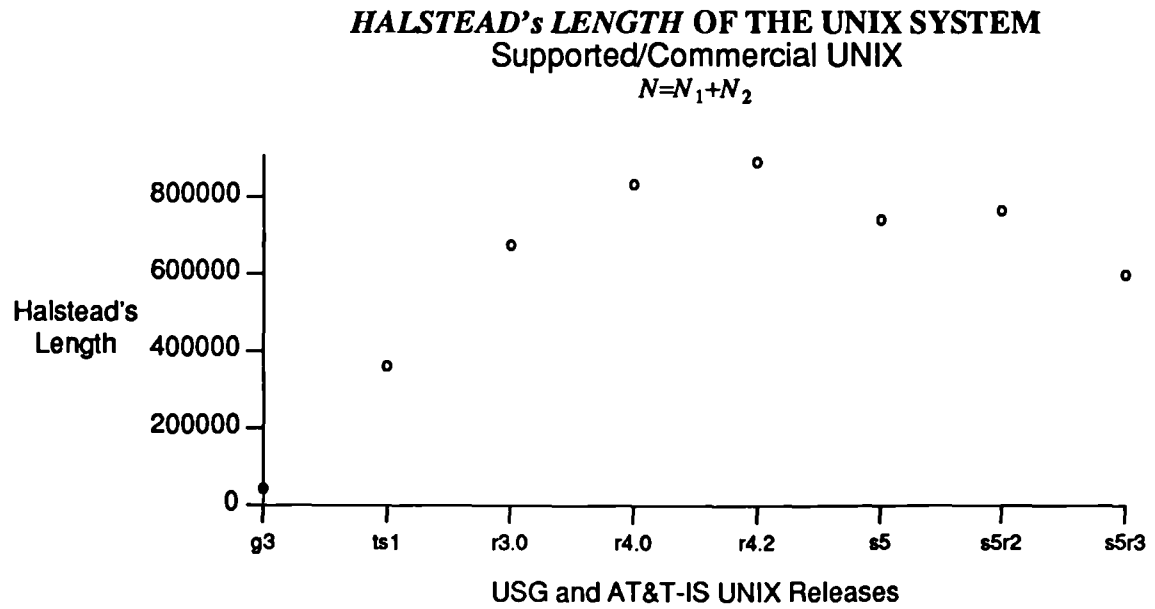


C.3 PLOTS REQUIRING THE PARSER

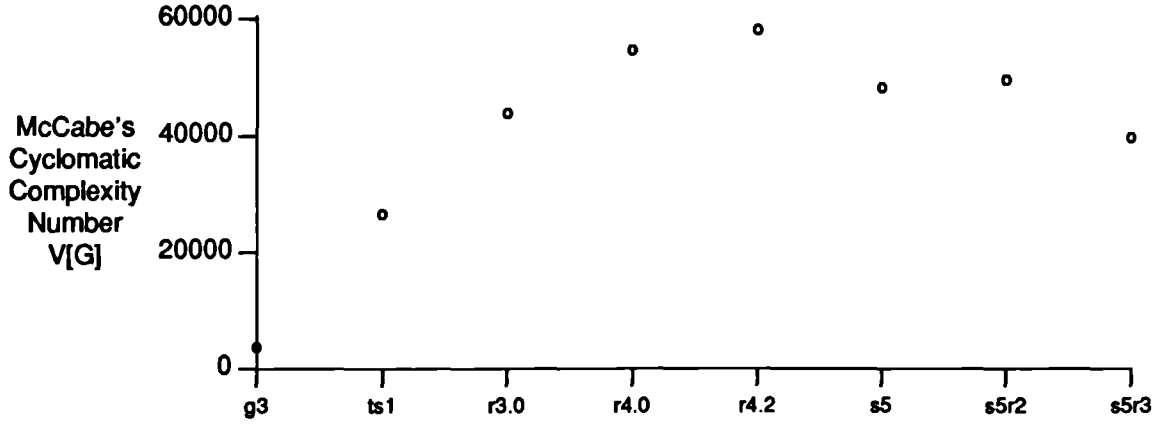
As explained in Chapter 4, all the metrics rejected for the plots required parsing the the 'C' source and due the inherent preprocessing problems not all the source files in the distribution tape could be successfully parsed, as shown below:



Therefore the following plots are of limited value since they are dominated by the parser's success. Halstead's *length*, McCabe's *cyclomatic complexity number* and the *number of function definitions* plots look suspiciously like the *number of statements* plot:

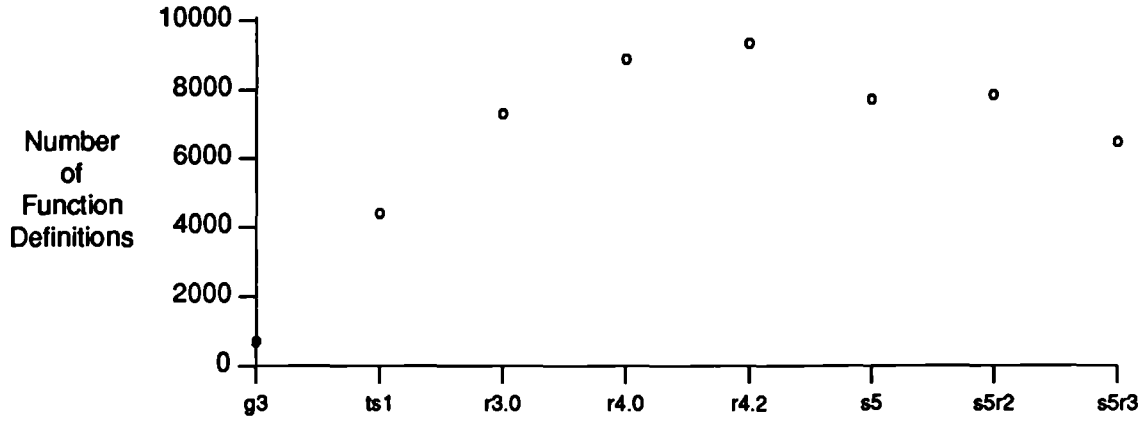


CYCLOMATIC COMPLEXITY OF THE UNIX SYSTEM
Supported/Commercial UNIX

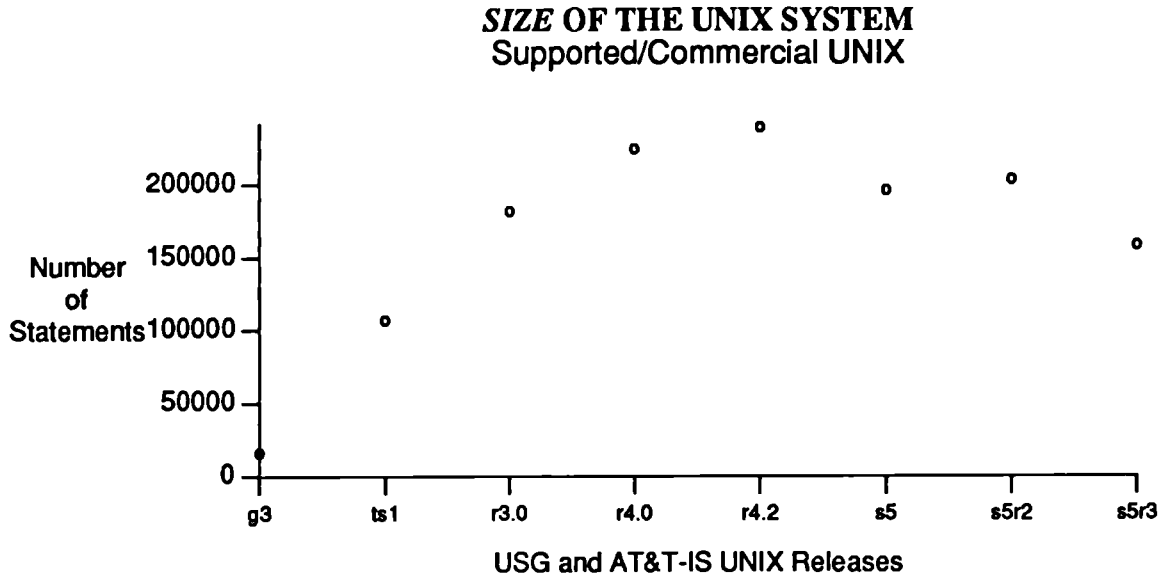


USG and AT&T-IS UNIX Releases

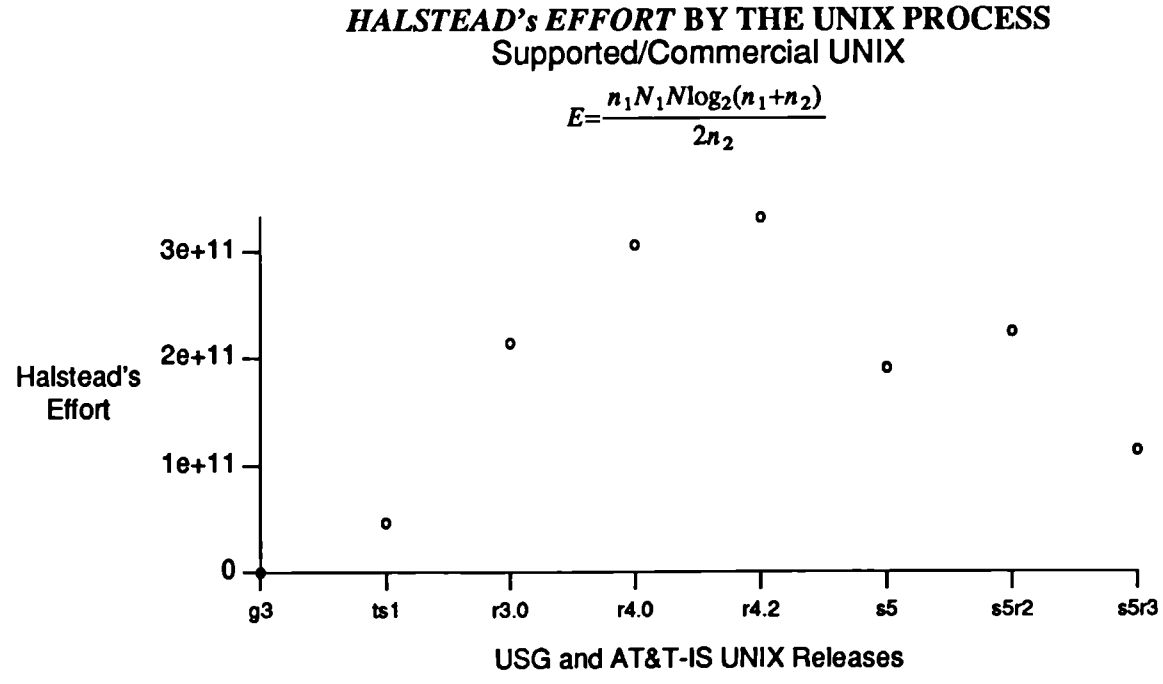
SIZE OF THE UNIX SYSTEM
Supported/Commercial UNIX



USG and AT&T-IS UNIX Releases



Halstead's metric for effort (claimed to be measuring the number of elementary mental discriminations required to understand the code) provides an exagerated (vertically) view of the *number of statements* plot:



The only metric which does not show the same pattern as size, and hence shows promise, is *number of external links*. This shows a clear increasing trend and implies that the number of external links *per file* has increased greatly during the past few releases. Several people (listed in Chapter 4) believe that this will make the code more difficult to understand.

Appendix D
PROGRAM LISTINGS

D.1 INTRODUCTION

This appendix contains listings of some of the many programs used in this investigation to arrive at the models presented in Chapter 5.

D.2 MECHANICS OF THE ANALYSIS

A number of steps were required to get from the mounted distribution to the graphs presented in the aforementioned chapter, described below.

D.2.1 Generating File List

Generate a list of all the required (mostly 'C' source) from the distribution. For this, the unix utility **find** was used:

*find . -name *.[ch] -type f -print*

assuming that the program was executed at the root of the mounted distribution and the output from **find** was redirected to a suitable file, say, *release_name.filelist*. Below is a taste of the contents of such a filelist:

```
src/uts/os/subr.c
src/uts/os/sys1.c
src/uts/os/sys2.c
src/uts/os/sys3.c
src/uts/os/sys4.c
src/uts/os/sysent.c
```

D.2.2 Analysing each file

Count various structure attributes in each file. Two programs were used for this: the standard UNIX utility **wc**; and the modified 'C' compiler, referred to in Chapter 4, **met** (for *met*rics generation). In both cases shell scripts were used to drive the programs:

wc 'cat release_name.filelist' > release_name.size

The number of files in some releases was too large for the **shell**[1] to handle so the filelist was appropriately split. The wc command counts the number of lines, words and characters in the given file(s). For example, a release_name.size file may look like:

1. The UNIX command interpreter.

```
src/uts/os/subr.c 195 538 3487
src/uts/os/sys1.c 518 1405 10002
src/uts/os/sys2.c 324 741 5097
src/uts/os/sys3.c 282 680 4844
src/uts/os/sys4.c 444 979 6422
src/uts/os/sysent.c 131 679 3172
```

Similarly **parser** was used to generate statistics on each file:

```
for i in 'cat release_name.filelist'
do
        dir_name='echo $i | sed 's:/[^/]*$::''
        file_name='echo $i | sed 's:^.*/::''
        > release_name.stats

        (cd $dir_name; parser file_name >> release.stats)
done
```

It had to be run for each file because it could not handle multiple files (unlike wc). The output generated by **parser** was of the form:

```
src/uts/os/subr.c  22 10 0 2 2 6 1 1 13 1 2 0 1 4 0 0 1 3 1 0 2 0 1 0 8 0 0 10 0 0 0 40 0 44 29 0 2 1 0 13 0 0
src/uts/os/sys1.c  107 20 0 8 45 35 18 10 45 15 4 0 10 3 0 0 11 1 0 4 4 0 0 0 33 8 8 6 6 4 2 288 0 216 139 4 1
src/uts/os/sys2.c  18 2 0 1 0 3 3 1 34 21 4 0 3 4 0 0 6 0 0 2 2 0 0 0 4 0 0 1 6 0 0 135 0 103 59 0 0 0 0 20 5
src/uts/os/sys3.c  41 9 0 1 3 4 7 3 21 10 6 0 2 1 0 0 1 0 0 2 2 0 0 0 5 3 6 3 5 0 1 151 0 106 80 0 3 5 0 19 9
src/uts/os/sys4.c  20 18 0 2 0 6 2 9 41 8 3 0 13 10 0 0 1 1 0 5 5 0 0 0 5 5 10 2 11 0 2 225 0 135 99 0 3 3 5 1
src/uts/os/sysent.c  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

Generating these two files (release_name.stats and release_name.size) consumed the most machine resources and took several hours to run on the larger releases.

D.2.3 Calculating metrics for the whole release

Awk programs were used to calculate the size and other metrics figures for the whole release. In most cases, the calculatin involved simply totalling the counts for each file in the release.

As, described in Chapter 4, calculating the changes since the previous release was slightly more complex as it involved knowing which files had changed names.

The **awk** programs to calculate the release totals presented below are given in the last section of this appendix.

Sample size.data file

```
g3   77 3 14     367 67005 170 18782
ts1   78 11 20     1262 192535 191 19312
r3.0   80 6 19     2120 341964 336 66874
r4.0   81 3 9     2546 412438 371 75669
r4.2   82 2 11     2874 513344 433 77834
s5   83 1 11     2129 373814 165 54522
s5r2   84 4 15     2422 406538 154 43315
s5r3   86 3 23     2815 596632 306 34749
```

Sample stats.data file

```
g3      268 99 16002 3703 1520 13567 11798 31630 695 12893
ts1      1104 158 107242 26548 9086 62925 111647 249100 4418 61482
r3.0      1870 240 181861 43936 17947 101683 212822 461638 7301 103985
r4.0      2258 278 225037 54536 20644 128593 266395 565899 8887 131809
r4.2      2446 472 239509 58006 21018 138519 283837 607168 9328 144004
s5      1890 239 196448 48157 16623 129548 232477 509876 7723 144035
s5r2      2221 191 203894 49618 19045 139354 247493 517908 7862 150780
s5r3      2101 711 158435 39778 16907 146519 190165 407980 6502 155080
```

D.2.4 Plotting the results

These plots were directly used to drive the UNIX text processing tools **grap, pic, eqn, tbl** and **troff** to produce the graphs and the rest of this thesis.

D.3 LISTINGS

D.3.1 Parser

For this investigation, mainly the parsing portions of the 'C' compiler were changed. this meant changing the files *scan.c* and *cgram.y*. Only the changed portion of *scan.c* is given since very little was changed.

```
/* @(#) scan.c: 1.2 2/27/84                              */

/*
**      Modified to count the number and size of comments
**
**      By Shamim Pirzada, Imperial College, London
**      April, 1986
**      Resident Visitor to Bell Labs Department 11271
**      Supervisor: Doug McIlroy
*/

# include "mfile1.h"
# include <ctype.h>
/* temporarily */

/* character-set translation */
# ifndef CCTRANS
# define CCTRANS(x) (x)
# endif


lxcom()
{
        extern int no_coms;
        extern int com_sz;
        register c;
        /* saw a /*: process a comment */

        ++no_coms;
        for (;;) {
                ++com_sz;
                switch (c = getchar()) {

                case EOF:
                        uerror( "lxcom() unexpected EOF" );
                        return;
```

```
                case '0:
                        ++lineno;

                default:
                        continue;

                case '*':
                        if( (c = getchar()) == '/' ) return;
                        else ungetchar();
                        continue;

                }
        }
}

#define Return(e)    return(yylval.lineno = lineno, (e))

yylex()
{
        for(;;)
        {

                register lxchar;
                register struct lxdope *p;
                register struct symtab *sp;
                register char *cp;
                register id;

                switch( (p = lxcp[(lxchar = getchar())+1])->lxact )
                {

onechar:
                        ungetc( lxchar );

                case A_1C:
                        /* eat up a single character, and return an opcode */

                        yylval.intval = p->lxval;
                        Return( p->lxtok );

                case A_ERR:
                        uerror( "illegal character: %03o (octal)", lxchar );
                        break;

                case A_LET:
                        /* collect an identifier, and check for reserved word */
                        lxget( lxchar, LEXLET|LEXDIG );

                        /* 0 means some kind of low level error, >0 reserved,
                        ** <0 means id
                        */
                        if( (lxchar=lxres()) > 0 ) Return( lxchar );
                        if( lxchar== 0 ) continue;

                if ((c = getchar()) != '0) {
                        cp = ftitle; *cp++ = c;
                        while ((c=getchar()) != '0)
                                *cp++ = c;
```

```
                    *cp = ' ';
                }
        }
}

# ifndef MYASMOUT
asmout()
{
#ifdef GDEBUG
        dbline();
#endif
        printx("%s151s151s0, ASM_COMMENT, asmbuf, ASM_END);
}
# endif
```

Since the yacc[2] input grammer was substantially changed to count various structures, *cgram.y* is given in its totality:

```
/* @(#) cgram.y: 1.2 2/9/84                          */
/*
**      Modified to generate statistics on occurances of function calls,
**      variables, constants, branches, loops
**      and function and variable declarations etc.
**
**      By Shamim Pirzada, Imperial College, London
**      April, 1986
**      Resident Visitor to Bell Labs Department 11271
**      Supervisor: Doug McIlroy
*/
%{
static char   SCCSID[] = "@(#) cgram.y:  2.1 83/08/02";
%}
%term NAME  2
%term STRING  3
%term ICON  4
%term FCON  5
%term PLUS   6
%term MINUS   8
%term MUL   11
%term AND   14
%term OR   17
%term ER   19
%term QUEST  21
%term COLON  22
%term ANDAND  23
%term OROR  24

/*      special interfaces for yacc alone */
/*      These serve as abbreviations of 2 or more ops:
        ASOP   =, = ops
        RELOP  LE,LT,GE,GT
        EQUOP  EQ,NE
        DIVOP  DIV,MOD
```

2. A UNIX utility to produce 'C' code from an input grammer. It was written to act as an aid in writing compilers.

```
        SHIFTOP       LS,RS
        ICOP    ICR,DECR
        UNOP    NOT,COMPL
        STROP   DOT,STREF

        */
%term ASOP  25
%term RELOP  26
%term EQUOP  27
%term DIVOP  28
%term SHIFTOP  29
%term INCOP  30
%term UNOP  31
%term STROP  32

/*      reserved words, etc */
%term TYPE  33
%term CLASS  34
%term STRUCT  35
%term RETURN  36
%term GOTO  37
%term IF  38
%term ELSE  39
%term SWITCH  40
%term BREAK  41
%term CONTINUE  42
%term WHILE  43
%term DO  44
%term FOR  45
%term DEFAULT  46
%term CASE  47
%term SIZEOF  48
%term ENUM 49

/*      little symbols, etc. */
/*      namely,

        LP      (
        RP      )

        LC      {
        RC      }

        LB      [
        RB      ]

        CM      ,
        SM      ;

        */

%term LP  50
%term RP  51
%term LC  52
%term RC  53
%term LB  54
%term RB  55
%term CM  56
```

```
%term SM  57
%term ASSIGN  58
%term ASM 59

/* at last count, there were 7 shift/reduce, 1 reduce/reduce conflicts
/* these involved:
       if/else
       recognizing functions in various contexts, including declarations
       error recovery
       */

%left CM
%right ASOP ASSIGN
%right QUEST COLON
%left OROR
%left ANDAND
%left OR
%left ER
%left AND
%left EQUOP
%left RELOP
%left SHIFTOP
%left PLUS MINUS
%left MUL DIVOP
%right UNOP
%right INCOP SIZEOF
%left LB LP STROP
%{
# include "mfile1.h"
%}

       /* define types */
%start ext_def_list

%type <intval> con_e ifelprefix ifprefix doprefix switchpart
              enum_head str_head name_lp
%type <nodep> e .e term attributes oattributes type enum_dcl struct_dcl
              cast_type null_decl funct_idn declarator fdeclarator
              nfdeclarator elist

%token <intval> CLASS NAME STRUCT RELOP CM DIVOP PLUS MINUS SHIFTOP MUL AND OR
              ER ANDAND OROR ASSIGN STROP INCOP UNOP ICON
%token <nodep> TYPE

%%

%{
       extern int wloop_level;    /* specifies while loop code generation */
       extern int floop_level;    /* specifies for loop code generation */
       static int fake = 0;
       static char fakename[NCHNAM+1];

       /*----------the count variables (by Shamim Pirzada, ssp)------*/
       int expr_stats, asign_stats, while_stats, for_stats, break_stats, /* diff */
           cont_stats, ret_stats, goto_stats, label_stats, case_stats,   /* statm*/
           deflt_stats, do_stats, if_stats, switch_stats, asm_stats;       /* types*/

       int optrs[32];                          /* count for operator occurances */
```

```
        int var_occ;                    /* count for variable occurances */
        int consts;                     /* count for constant occurances */
        int func_calls;                       /* count for function calls      */
        int e_func_calls;               /* count for extern func calls   */
        int no_coms;                    /* no of comments           */
        int com_sz;                     /* total comment space (in chars)*/
        int var_decls;                        /* no of vars declared */
        int func_decls;                       /* no of functions declared */
        int tl_defs;                    /* total data defs (var + funcs) */
        int func_defs;                        /* function definitions */
        int scope;
        int extern_vars;
        int params;
        int i_var_decls;
        int ref_decls;
        int operands;

%}

ext_def_list:   ext_def_list external_def
              |     /* EMPTY */
                    {
#ifndef LINT
                    beg_file();
#endif
                     ftnend();
                     }
              ;
external_def:   data_def
                    { curclass = SNULL;  blevel = 0; }
              | error
                    { curclass = SNULL;  blevel = 0; }
              | ASM SM
                    { asmout();  curclass = SNULL; blevel = 0; }
              ;
data_def:       oattributes  SM
                    {  $1->in.op = FREE; }
              | oattributes init_dcl_list  SM
                    {  $1->in.op = FREE; }
              | oattributes fdeclarator
                    {   defid( tymerge($1,$2),
                                curclass==STATIC?STATIC:EXTDEF ); }
                    function_body
                    {
                        if( blevel ) cerror( "function level error" );
                        if( reached ) retstat |= NRETVAL;
                        $1->in.op = FREE;
                        ++func_defs;
                        ftnend();
                        }
              ;

function_body:          arg_dcl_list compoundstmt
              {     regvar = 0; }  /* clear out arguments */
              ;

arg_dcl_list:   arg_dcl_list attributes declarator_list SM
                    { curclass = SNULL;  $2->in.op = FREE; }
```

```
            |  arg_dcl_list attributes SM
                   /* to permit structs in decl. lists */
                   { curclass = SNULL;  $2->in.op = FREE; }
            |      /* EMPTY */  {  blevel = 1; }
            ;

stmt_list:     stmt_list statement
            |  /* EMPTY */
                   {  bccode();
                      locctr(PROG);
                      }
            ;

dcl_stat_list:  dcl_stat_list attributes SM
                   {  $2->in.op = FREE; }
            |  dcl_stat_list attributes init_dcl_list SM
                   {  $2->in.op = FREE; }
            |  /* EMPTY */
            ;

oattributes:   attributes
            |  /* empty */
                   {  $$ = mkty(INT,0,INT);  curclass = SNULL; }
            ;
attributes:    class type
                   {  $$ = $2; }
            |  type class
            |  class
                   {  $$ = mkty(INT,0,INT); }
            |  type
                   { curclass = SNULL ; }
            |  type class type
                   {  $1->in.type = types( $1->in.type, $3->in.type,
                              UNDEF );
                      $3->in.op = FREE;
                      }
            ;

class:         CLASS
                   {  curclass = $1; }
            ;

type:          TYPE
            |  TYPE TYPE
                   {  $1->in.type = types( $1->in.type, $2->in.type, UNDEF );
                      $2->in.op = FREE;
                      }
            |  TYPE TYPE TYPE
                   {  $1->in.type = types( $1->in.type, $2->in.type, $3->in.type );
                      $2->in.op = $3->in.op = FREE;
                      }
            |  struct_dcl
            |  enum_dcl
            ;

enum_dcl:      enum_head LC moe_list optcomma RC
                   { $$ = dclstruct($1); }
            |  ENUM NAME
```

```
                { $$ = rstruct($2,0);  stwart = instruct; }
        ;

enum_head:      ENUM
                { $$ = bstruct(-1,0); stwart = SEENAME; }
        |       ENUM NAME
                { $$ = bstruct($2,0); stwart = SEENAME; }
        ;

moe_list:       moe
        |       moe_list CM moe
        ;

moe:            NAME
                { moedef( $1 ); }
        |       NAME ASSIGN con_e
                { strucoff = $3;  moedef( $1 ); }
        ;

struct_dcl:     str_head LC type_dcl_list optsemi RC
                { $$ = dclstruct($1);  }
        |       STRUCT NAME
                { $$ = rstruct($2,$1); }
        ;

str_head:       STRUCT
                { $$ = bstruct(-1,$1);  stwart=0; }
        |       STRUCT NAME
                { $$ = bstruct($2,$1);  stwart=0;  }
        ;

type_dcl_list:      type_declaration
        |       type_dcl_list SM type_declaration
        ;

type_declaration:  type declarator_list
                { curclass = SNULL;  stwart=0; $1->in.op = FREE; }
        |       type
                { if( curclass != MOU ){
                      curclass = SNULL;
                      }
                   else {
                      sprintf( fakename, "$%dFAKE", fake++ );
                      defid( tymerge($1, bdty(NAME,NIL,
                             lookup( fakename, SMOS ))), curclass );
                      werror("union member must be named");
                      }
                   stwart = 0;
                   $1->in.op = FREE;
                   }
        ;


declarator_list:   declarator
                { defid( tymerge($<nodep>0,$1), curclass);
                      stwart = instruct;
                }
        |       declarator_list  CM {$<nodep>$=$<nodep>0;}  declarator
```

```
                { defid( tymerge($<nodep>0,$4), curclass);
                      stwart = instruct;
                }
        ;
declarator:     nfdeclarator
        |       nfdeclarator COLON con_e
                %prec CM
                {  if( !(instruct&INSTRUCT) )
                      uerror( "field outside of structure" );
                   if( $3<0 || $3 >= FIELD ){
                      uerror( "illegal field size" );
                      $3 = 1;
                      }
                   defid( tymerge($<nodep>0,$1), FIELD|$3 );
                   $$ = NIL;
                   }
        |       COLON con_e
                %prec CM
                {   if( !(instruct&INSTRUCT) )
                      uerror( "field outside of structure" );
                   /* alignment or hole */
                   falloc( stab, $2, -1, $<nodep>0 );
                   $$ = NIL;
                   }
        |       error
                {  $$ = NIL; }
        ;

        /* int (a)();   is not a function --- sorry! */
nfdeclarator:   MUL nfdeclarator
                {  umul:
                      $$ = bdty( UNARY MUL, $2, 0 ); }
        |       nfdeclarator  LP   RP
                {  uftn:
                      $$ = bdty( UNARY CALL, $1, 0 );  }
        |       nfdeclarator LB RB
                {  uary:
                      $$ = bdty( LB, $1, 0 );  }
        |       nfdeclarator LB con_e RB
                {  bary:
                      if( (int)$3 <= 0 )
                            werror( "zero or negative subscript" );
                      $$ = bdty( LB, $1, $3 );  }
        |       NAME
                {  $$ = bdty( NAME, NIL, $1 );  }
        |        LP  nfdeclarator  RP
                { $$=$2; }
        ;
fdeclarator:    MUL fdeclarator
                {  goto umul; }
        |       fdeclarator  LP   RP
                {  goto uftn; }
        |       fdeclarator LB RB
                {  goto uary; }
        |       fdeclarator LB con_e RB
                {  goto bary; }
        |        LP  fdeclarator  RP
                { $$ = $2; }
```

```
            |  name_lp
                { if (paramno)
                      uerror("arg list in declaration"); }
                name_list  RP
                {
                      if( blevel!=0 )
                            uerror(
                            "function declaration in bad context");
                      $$ = bdty( UNARY CALL, bdty(NAME,NIL,$1), 0 );
                      stwart = 0;
                      }
            |  name_lp
                { if (paramno)
                      uerror("arg list in declaration"); }
                RP
                {
                      $$ = bdty( UNARY CALL, bdty(NAME,NIL,$1), 0 );
                      stwart = 0;
                      }
            ;

name_lp:      NAME LP
                {
                      /* turn off typedefs for argument names */
                      stwart = SEENAME;
                      if( stab[$1].sclass == SNULL )
                          stab[$1].stype = FTN;
                      }
            ;

name_list:     NAME
                { ftnarg( $1 );  stwart = SEENAME;
                  ++params;
                }
            |  name_list  CM  NAME
                { ftnarg( $3 );  stwart = SEENAME;
                  ++params;
                }
            | error
            ;
            /* always preceeded by attributes: thus the $<nodep>0's */
init_dcl_list:       init_declarator
                %prec CM
                {     ++ref_decls;
                }
            |  init_dcl_list  CM {$<nodep>$=$<nodep>0;}  init_declarator
                {     ++ref_decls;
                }
            ;
            /* always preceeded by attributes */
xnfdeclarator:       nfdeclarator
                {  defid( $1 = tymerge($<nodep>0,$1), curclass);
                    beginit($1->tn.rval);
                    }
            |  error
            ;
            /* always preceeded by attributes */
init_declarator:   nfdeclarator
```

```
                    {  nidcl( tymerge($<nodep>0,$1) );
                           switch(blevel){
                           case 0:      ++extern_vars; break;
                           default:     ++i_var_decls; break;
                           }
                    }
               |  fdeclarator
                    {  defid( tymerge($<nodep>0,$1), uclass(curclass) );
                    }
               |  xnfdeclarator ASSIGN e
                    %prec CM
                    {  doinit( $3 );
                           switch(blevel){
                           case 0:      ++extern_vars; break;
                           default:     ++i_var_decls; break;
                           }
                        endinit(); }
               |  xnfdeclarator ASSIGN LC init_list optcomma RC
                    {  endinit();
                           switch(blevel){
                           case 0:      ++extern_vars; break;
                           default:     ++i_var_decls; break;
                           }
                    }
               | error
               ;

init_list:       initializer
                    %prec CM
               |  init_list  CM  initializer
               ;
initializer:     e
                    %prec CM
                    {  doinit( $1 ); }
               |  ibrace init_list optcomma RC
                    {  irbrace(); }
               ;

optcomma       :     /* empty */
               |  CM
               ;

optsemi              :     /* empty */
               |  SM
               ;

ibrace         : LC
                    {  ilbrace(); }
               ;

/*     STATEMENTS    */

compoundstmt:    begin dcl_stat_list stmt_list RC
                    {
                        clearst(blevel);
                        if (--blevel == 1)
                        {
                           clearst(blevel);
```

```
                        blevel = 0;
                    }
                    checkst(blevel);
                    autooff = *--psavbc;
                    regvar = *--psavbc;
                }
        ;

begin:      LC
                {  if( blevel == 1 ) dclargs();
                    uplevel();
                    if( psavbc > &asavbc[BCSZ-2] )
                            cerror( "nesting too deep" );
                    *psavbc++ = regvar;
                    *psavbc++ = autooff;
                    }
        ;

statement:      e SM
                    { ecomp($1);
                      ++expr_stats;
                    }
            |   ASM SM
                    { asmout();
                      ++asm_stats;
                    }
            |   compoundstmt
            |   ifprefix statement
                    { deflab($1);
                      reached = 1;
#ifdef M32B
                      brdepth--;
#endif
                      }
            |   ifelprefix statement
                    {  if( $1 != NOLAB ){
                           deflab( $1 );
                           reached = 1;
                           }
#ifdef M32B
                       brdepth--;
#endif
                       }
            |   WHILE
                    {
                    ++while_stats;
#ifdef M32B
                          whdepth++;
#endif
                    }
                    LP e RP
                    {
                          savebc();
                          if (!reached)
                                werror("loop not entered at top");
                          reached = 1;
                          brklab = getlab();
                          contlab = getlab();
```

```
                switch (wloop_level) {
                default:
                        cerror("bad while loop code gen value");
                        /*NOTREACHED*/
                case LL_TOP: /* test at loop top */
                        deflab(contlab);
                        if ($4->in.op == ICON && $4->tn.lval) {
                                flostat = FLOOP;
                                tfree($4);
                        } else {
                                $4->ln.lineno = $<lineno>1;
                                ecomp(buildtree(CBRANCH, $4,
                                    bcon(brklab)));
                        }
                        break;
                case LL_BOT: /* test at loop bottom */
                        if ($4->in.op == ICON && $4->tn.lval) {
                                flostat = FLOOP;
                                tfree($4);
                                deflab(contlab);
                        } else {
                                branch(contlab);
                                deflab($<intval>$ = getlab());
                        }
                        break;
                case LL_DUP: /* dup. test at top & bottom */
                        if ($4->in.op == ICON && $4->tn.lval) {
                                flostat = FLOOP;
                                tfree($4);
                                deflab($<intval>$ = contlab);
                        } else {
                                register NODE *sav;
                                extern NODE *treecpy();
                                sav = treecpy($4);
                                ecomp(buildtree(CBRANCH,$4,
                                    bcon(brklab)));
                                $4 = sav;
                                deflab($<intval>$ = getlab());
                        }
                        break;
                }
        }
        statement
{
        switch (wloop_level) {
        default:
                cerror("bad while loop code gen. value");
                /*NOTREACHED*/
        case LL_TOP: /* test at loop top */
                branch(contlab);
                break;
        case LL_BOT: /* test at loop bottom */
                if (flostat & FLOOP)
                        branch(contlab);
                else {
                        reached = 1;
                        deflab(contlab);
                        ecomp(buildtree(CBRANCH,
```

```
                                        buildtree(NOT, $4, NIL),
                                        bcon($<intval>6)));
                        }
                        break;
                case LL_DUP: /* dup. test at top & bottom */
                        if (flostat & FLOOP)
                                branch(contlab);
                        else {
                                if (flostat & FCONT) {
                                        reached = 1;
                                        deflab(contlab);
                                }
                                ecomp(buildtree(CBRANCH,
                                        buildtree(NOT, $4, NIL),
                                        bcon($<intval>6)));
                        }
                        break;
                }
                if ((flostat & FBRK) || !(flostat & FLOOP))
                        reached = 1;
                else
                        reached = 0;
                deflab(brklab);
                resetbc(0);
#ifdef M32B
                whdepth--;
#endif
        }
        | doprefix statement WHILE LP e RP SM
            {   deflab( contlab );
                if( flostat & FCONT ) reached = 1;
                $5->ln.lineno = $<lineno>3;
                ecomp( buildtree(CBRANCH,
                   buildtree(NOT, $5, NIL), bcon($1)));
                deflab( brklab );
                reached = 1;
                resetbc(0);
#ifdef M32B
                whdepth--;
#endif
                }
        | FOR LP .e SM
            {
            ++for_stats;
#ifdef M32B
                        fordepth++;
#endif
                }
                .e SM
                {
                        if ($3) {
                                $3->ln.lineno = $<lineno>1;
                                ecomp($3);
                        } else if (!reached)
                                werror("loop not entered at top");
                        savebc();
                        contlab = getlab();
                        brklab = getlab();
```

```
                reached = 1;
                switch (floop_level) {
                default:
                        cerror("bad for loop code gen. value");
                        /*NOTREACHED*/
                case LL_TOP: /* test at loop top */
                        deflab($<intval>$ = getlab());
                        if (!$6)
                                flostat |= FLOOP;
                        else if ($6->in.op == ICON && $6->tn.lval) {
                                flostat |= FLOOP;
                                tfree($6);
                                $6 = (NODE *)0;
                        } else {
                                if (!$3)
                                        $6->ln.lineno = $<lineno>1;
                                ecomp(buildtree(CBRANCH, $6,
                                    bcon(brklab)));
                        }
                        break;
                case LL_BOT: /* test at loop bottom */
                        if (!$6)
                                flostat |= FLOOP;
                        else if ($6->in.op == ICON && $6->tn.lval) {
                                flostat |= FLOOP;
                                tfree($6);
                                $6 = (NODE *)0;
                        } else
                                branch($<intval>1 = getlab());
                        deflab($<intval>$ = getlab());
                        break;
                case LL_DUP: /* dup. test at top & bottom */
                        if (!$6)
                                flostat |= FLOOP;
                        else if ($6->in.op == ICON && $6->tn.lval) {
                                flostat |= FLOOP;
                                tfree($6);
                                $6 = (NODE *)0;
                        } else {
                                register NODE *sav;
                                extern NODE *treecpy();
                                sav = treecpy($6);
                                ecomp(buildtree(CBRANCH, $6,
                                        bcon(brklab)));
                                $6 = sav;
                        }
                        deflab($<intval>$ = getlab());
                        break;
                }
        }
        .e RP statement
{
        if (flostat & FCONT) {
                deflab(contlab);
                reached = 1;
        }
        if ($9)
                $9->ln.lineno = lineno, ecomp($9);
```

```
                    switch (floop_level) {
                    default:
                          cerror("bad for loop code gen. value");
                          /*NOTREACHED*/
                    case LL_TOP: /* test at loop top */
                          branch($<intval>8);
                          break;
                    case LL_BOT: /* test at loop bottom */
                          if ($6)
                                deflab($<intval>1);
                          /*FALLTHROUGH*/
                    case LL_DUP: /* dup. test at top & bottom */
                          if ($6) {
                                ecomp(buildtree(CBRANCH,
                                      buildtree(NOT, $6, NIL),
                                      bcon($<intval>8)));
                          } else
                                branch($<intval>8);
                          break;
                    }
                    deflab(brklab);
                    if ((flostat & FBRK) || !(flostat & FLOOP))
                          reached = 1;
                    else
                          reached = 0;
                    resetbc(0);
#ifdef M32B
                    fordepth--;
#endif
              }
              | switchpart statement
                    {   if( reached ) branch( brklab );
                        deflab( $1 );
                        swend();
                        deflab(brklab);
                        if( (flostat&FBRK) || !(flostat&FDEF) ) reached=1;
                        resetbc(FCONT);
#ifdef M32B
                        brdepth--;
#endif
                        }
              |  BREAK  SM
                    {   ++break_stats;
                        if( brklab == NOLAB ) uerror( "illegal break");
                        else if(reached) {
                           slineno = $<lineno>1; dbline();
                           branch( brklab );
                        }
                        flostat |= FBRK;
                        if( brkflag ) goto rch;
                        reached = 0;
                        }
              |  CONTINUE  SM
                    {   ++cont_stats;
                        if( contlab == NOLAB )
                           uerror( "illegal continue");
                        else {
                           slineno = $<lineno>1; dbline();
```

```
                    branch( contlab );
                }
                flostat |= FCONT;
                goto rch;
            }
        |   RETURN  SM
            {   ++ret_stats;
                retstat |= NRETVAL;
                slineno = $<lineno>1; dbline();
                branch( retlab );
            rch:
                if( !reached ) werror( "statement not reached");
                reached = 0;
            }
        |   RETURN e  SM
            {   register NODE *temp;
                TWORD indtype();

                ++ret_stats;
                idname = curftn;
                temp = buildtree( NAME, NIL, NIL );
                temp->in.type = DECREF(temp->in.type);
                if(temp->in.type == (FTN|VOID))
                   uerror(
                   "void function %s cannot return value",
                          stab[idname].sname);
                temp->tn.op = RNODE;
                $2 = makety( $2, temp->fn.type,
                          temp->fn.cdim, temp->fn.csiz );
                temp->in.type = indtype( temp->in.type );
                temp = buildtree( ASSIGN, temp, $2 );
                temp->ln.lineno = $<lineno>1;
                ecomp( temp );
                retstat |= RETVAL;
                branch( retlab );
                reached = 0;
            }
        |   GOTO NAME SM
            {  register NODE *q;
                ++goto_stats;
                q = block( FREE, NIL, NIL, INT|ARY, 0, INT );
                q->tn.rval = idname = $2;
                defid( q, ULABEL );
                stab[idname].suse = -lineno;
                slineno = $<lineno>1; dbline();
                branch( stab[idname].offset );
                goto rch;
                }
        |    SM
        |   error  SM
        |   error RC
        |   label statement
        ;
label:      NAME COLON
            {  register NODE *q;
                ++label_stats;
                q = block( FREE, NIL, NIL, INT|ARY, 0, LABEL );
                q->tn.rval = $1;
```

```
                    defid( q, LABEL );
                    reached = 1;
                    }
            |  CASE e COLON
                { ++case_stats;
                    addcase($2);
                    reached = 1;
                    }
            |  DEFAULT COLON
                { ++deflt_stats;
                    reached = 1;
                    adddef();
                    flostat |= FDEF;
                    }
            ;
doprefix:   DO
                { ++do_stats;
                    savebc();
                    if( !reached ) werror( "loop not entered at top");
                    brklab = getlab();
                    contlab = getlab();
                    deflab( $$ = getlab() );
                    reached = 1;
#ifdef M32B
                    whdepth++;
#endif
                    }
            ;
ifprefix:   IF LP e RP
                {   ++if_stats;
                    $3->ln.lineno = $<lineno>1;
                    ecomp(buildtree(CBRANCH, $3, bcon($$=getlab()))) ;
                    reached = 1;
#ifdef M32B
                    brdepth++;
#endif
                    }
            ;
ifelprefix:   ifprefix statement ELSE
                { if( reached ) branch( $$ = getlab() );
                    else $$ = NOLAB;
                    deflab( $1 );
                    reached = 1;
                    }
            ;

switchpart:     SWITCH  LP  e  RP
                {   register NODE *temp;
                    ++switch_stats;
                    savebc();
                    temp = block( SNODE, NIL, NIL, INT, 0, INT );
                    temp = buildtree( ASSIGN, temp, $3 );
#ifdef M32B
                    temp = setswreg( temp );
#endif
                    brklab = getlab();
                    temp->ln.lineno = $<lineno>1;
                    ecomp( temp );
```

```
                        branch( $$ = getlab() );
                        swstart();
                        reached = 0;
#ifdef M32B
                        brdepth++;
#endif
                        }
                ;
/*      EXPRESSIONS     */
con_e:          { $<intval>$=instruct; stwart=instruct=0; } e
                    %prec CM
                    {  $$ = icons( $2 ); instruct=$<intval>1; }
                ;
                |
                    { $$ = 0; }
                ;
elist:          e
                    %prec CM
                |   elist  CM  e
                    {  goto bop; }
                ;

e:              e RELOP e
                    {
                        ++optrs[1];
                    preconf:
                        if( yychar==RELOP || yychar==EQUOP || yychar==AND
                                || yychar==OR || yychar==ER ){
                    precplaint:
                            if( hflag ) werror(
                            "precedence confusion possible: parenthesize!"
                                );
                            }
                    bop:
                        $$ = buildtree( $2, $1, $3 );
                        }
                |   e CM e
                        {       ++optrs[2];
                                $2 = COMOP;
                                goto bop;
                        }
                |   e DIVOP e
                        {       ++optrs[3];
                                goto bop;
                        }
                |   e PLUS e
                        {       ++optrs[4];
                                if(yychar==SHIFTOP)
                                        goto precplaint;
                                else
                                        goto bop;
                        }
                |   e MINUS e
                        {       ++optrs[5];
                                if(yychar==SHIFTOP )
                                        goto precplaint;
                                else
                                        goto bop;
```

```
        }
|   e SHIFTOP e
        {       ++optrs[6];
                if(yychar==PLUS||yychar==MINUS)
                        goto precplaint;
                else
                        goto bop;
        }
|   e MUL e
        {       ++optrs[7];
                goto bop;
        }
|   e EQUOP  e
        {       ++optrs[8];
                goto preconf;
        }
|   e AND e
        {       ++optrs[9];
                if( yychar==RELOP||yychar==EQUOP )
                        goto preconf;
                else
                        goto bop;
        }
|   e OR e
        {       ++optrs[10];
                if(yychar==RELOP||yychar==EQUOP)
                        goto preconf;
                else
                        goto bop;
        }
|   e ER e
        {       ++optrs[11];
                if(yychar==RELOP||yychar==EQUOP)
                        goto preconf;
                else
                        goto bop;
        }
|   e ANDAND e
        {       ++optrs[12];
                goto bop;
        }
|   e OROR e
        {       ++optrs[13];
                goto bop;
        }
|   e MUL ASSIGN e
        {       ++optrs[14];
        abop:
                $$ = buildtree( ASG $2, $1, $4 );
                }
|   e DIVOP ASSIGN e
        {       ++optrs[15];
                goto abop;
        }
|   e PLUS ASSIGN e
        {       ++optrs[16];
                goto abop;
        }
```

```
                |  e MINUS ASSIGN e
                       {       ++optrs[17];
                               goto abop;
                       }
                |  e SHIFTOP ASSIGN e
                       {       ++optrs[18];
                               goto abop;
                       }
                |  e AND ASSIGN e
                       {       ++optrs[19];
                               goto abop;
                       }
                |  e OR ASSIGN e
                       {       ++optrs[20];
                               goto abop;
                       }
                |  e ER ASSIGN e
                       {       ++optrs[21];
                               goto abop;
                       }
                |  e QUEST e COLON e
                       {       ++optrs[22];
                               $$=buildtree(QUEST, $1, buildtree( COLON, $3, $5 ) );
                       }
                |  e ASOP e
                       {       ++optrs[23];
                               werror( "old-fashioned assignment operator" );
                               goto bop;
                       }
                |  e ASSIGN e
                       {       ++asign_stats;
                               goto bop;
                       }
                |  term
                       {       ++operands;
                       }
                ;

term:           term INCOP
                       {       ++optrs[24];
                               $$ = buildtree( $2, $1, bcon(1) );
                       }
                |  MUL term
                       {       ++optrs[25];
                              ubop:
                                 $$ = buildtree( UNARY $1, $2, NIL );
                       }
                |  AND term
                       {       ++optrs[26];
#ifdef M32B
                               myand($2);
#endif
                               if( ISFTN($2->in.type) || ISARY($2->in.type) ){
                                      werror( "& before array or function: ignored" );
                                      $$ = $2;
                               } else
                                      goto ubop;
                        }
```

```
| MINUS term
    {       ++optrs[27];
            goto ubop;
    }
| UNOP term
    {       ++optrs[28];
            $$ = buildtree( $1, $2, NIL );
    }
| INCOP term
    {       ++optrs[29];
            $$ = buildtree( $1==INCR ? ASG PLUS : ASG MINUS,
                        $2,
                        bcon(1)  );
    }
| SIZEOF term
    {       ++optrs[30];
            $$ = doszof( $2 );
    }
| LP cast_type RP term  %prec INCOP
    {  $$ = buildtree( CAST, $2, $4 );
       $$->in.left->in.op = FREE;
       $$->in.op = FREE;
       $$ = $$->in.right;
        }
| SIZEOF LP cast_type RP  %prec SIZEOF
    {  $$ = doszof( $3 ); }
| term LB e RB
    {  $$ = buildtree( LB, $1, $3 ); }
| term LB e COLON e RB
    {  $$ = xicolon( $1, $3, $5 ); }
| funct_idn  RP
    {  $$=buildtree(UNARY CALL,$1,NIL);
    }
| funct_idn elist  RP
    {  $$=buildtree(CALL,$1,$2); }
| term STROP NAME
    {  ++var_occ;
       ++optrs[31];
       if( $2 == DOT ){
           if( notlval( $1 ) )werror(
                   "structure reference must be addressable"
                   );
           $1 = buildtree( UNARY AND, $1, NIL );
           }
        idname = $3;
        $$ = buildtree( STREF, $1,
                   buildtree( NAME, NIL, NIL ) );
        }
| NAME
    {  ++var_occ;
        idname = $1;
        /* recognize identifiers in initializations */
        if( blevel==0 && stab[idname].stype == UNDEF ) {
           register NODE *q;
           werror( "undeclared initializer name %s",
                   stab[idname].sname );
           q = block( FREE, NIL, NIL, INT, 0, INT );
           q->tn.rval = idname;
```

```
                        defid( q, EXTERN );
                        }
                    $$=buildtree(NAME,NIL,NIL);
                    stab[$1].suse = -lineno;
                }
            |   ICON
                {   ++consts;
                    $$=bcon(0);
                    $$->tn.lval = lastcon;
                    $$->tn.rval = NONAME;
                    if( $1 ) $$->fn.csiz = $$->in.type = ctype(LONG);
                }
            |   FCON
                {   ++consts;
                    $$=buildtree(FCON,NIL,NIL);
                    $$->fpn.dval = dcon;
                }
            |   STRING
                {   ++consts;
                    $$ = getstr();
                }
            |    LP   e   RP
                {   ++optrs[0];
                    $$=$2;
                }
            ;

cast_type:      type null_decl
                {
                $$ = tymerge( $1, $2 );
                $$->in.op = NAME;
                $1->in.op = FREE;
                }
            ;

null_decl:      /* EMPTY */
                { $$ = bdty( NAME, NIL, -1 ); }
            |   LP RP
                { $$ = bdty( UNARY CALL, bdty(NAME,NIL,-1),0); }
            |   LP null_decl RP LP RP
                {   $$ = bdty( UNARY CALL, $2, 0 ); }
            |   MUL null_decl
                {   goto umul; }
            |   null_decl LB RB
                {   goto uary; }
            |   null_decl LB con_e RB
                {   goto bary;   }
            |   LP null_decl RP
                { $$ = $2; }
            ;

funct_idn:      NAME   LP
                {       ++func_calls;
                        if( stab[$1].stype == UNDEF ){
                                register NODE *q;
                                q = block( FREE, NIL, NIL, FTN|INT, 0, INT );
                                q->tn.rval = $1;
                                defid( q, EXTERN );
```

```
                                ++e_func_calls;
                        }
                        idname = $1;
                        $$=buildtree(NAME,NIL,NIL);
                        stab[idname].suse = -lineno;
                }
        |  term  LP
                {       ++func_calls;
                }
        ;
%%

NODE *
mkty( t, d, s ) unsigned t; {
        return( block( TYPE, NIL, NIL, t, d, s ) );
        }

NODE *
bdty( op, p, v ) NODE *p; {
        register NODE *q;

        q = block( op, p, NIL, INT, 0, INT );

        switch( op ){

        case UNARY MUL:
        case UNARY CALL:
                break;

        case LB:
                q->in.right = bcon(v);
                break;

        case NAME:
                q->tn.rval = v;
                break;

        default:
                cerror( "bad bdty" );
                }

        return( q );
        }

dstash( n ){ /* put n into the dimension table */
        if( curdim >= DIMTABSZ-1 ){
                cerror( "dimension table overflow");
                }
        dimtab[ curdim++ ] = n;
        }

savebc() {
        if( psavbc > & asavbc[BCSZ-4 ] ){
                cerror( "whiles, fors, etc. too deeply nested");
                }
        *psavbc++ = brklab;
        *psavbc++ = contlab;
        *psavbc++ = flostat;
```

```
        *psavbc++ = swx;
#ifdef M32B
        *psavbc++ = swregno;
#endif
        flostat = 0;
        }

resetbc(mask){

#ifdef M32B
        swregno = *--psavbc;
#endif
        swx = *--psavbc;
        flostat = *--psavbc | (flostat&mask);
        contlab = *--psavbc;
        brklab = *--psavbc;

        }

addcase(p) NODE *p; { /* add case to switch */

        p = optim( p );  /* change enum to ints */
        if( p->in.op != ICON ){
                uerror( "non-constant case expression");
                return;
                }
        if( swp == swtab ){
                uerror( "case not in switch");
                return;
                }
        if( swp >= &swtab[SWITSZ] ){
                cerror( "switch table overflow");
                }
        swp->sval = p->tn.lval;
        deflab( swp->slab = getlab() );
        ++swp;
        tfree(p);
        }

adddef(){ /* add default case to switch */
        if( swtab[swx].slab >= 0 ){
                uerror( "duplicate default in switch");
                return;
                }
        if( swp == swtab ){
                uerror( "default not inside switch");
                return;
                }
        deflab( swtab[swx].slab = getlab() );
        }

swstart(){
        /* begin a switch block */
        if( swp >= &swtab[SWITSZ] ){
                cerror( "switch table overflow");
                }
        swx = swp - swtab;
        swp->slab = -1;
```

```
        ++swp;
        }

swend(){ /* end a switch block */

        register struct sw *swbeg, *p, *q, *r, *r1;
        CONSZ temp;
        int tempi;

        swbeg = &swtab[swx+1];

        /* sort */

        r1 = swbeg;
        r = swp-1;

        while( swbeg < r ){
                /* bubble largest to end */
                for( q=swbeg; q<r; ++q ){
                        if( q->sval > (q+1)->sval ){
                                /* swap */
                                r1 = q+1;
                                temp = q->sval;
                                q->sval = r1->sval;
                                r1->sval = temp;
                                tempi = q->slab;
                                q->slab = r1->slab;
                                r1->slab = tempi;
                                }
                        }
                r = r1;
                r1 = swbeg;
                }

        /* it is now sorted */

        for( p = swbeg+1; p<swp; ++p ){
                if( p->sval == (p-1)->sval ){
                        uerror( "duplicate case in switch, %d", tempi=p->sval );
                        return;
                        }
                }

        reached = 1;
        genswitch( swbeg-1, (int)(swp-swbeg) );
        swp = swbeg-1;
        }


finish() {
int i,j,k;
extern int optrs[];                     /* count for the 32 operator occ */
extern int expr_stats, asign_stats, while_stats, for_stats, break_stats, /* diff */
          cont_stats, ret_stats, goto_stats, label_stats, case_stats,   /* statm*/
          deflt_stats, do_stats, if_stats, switch_stats,asm_stats; /* types*/
extern int var_occ;                     /* count for variable occurances */
extern int consts;                      /* count for constant occurances */
extern int func_calls;                          /* count for function calls      */
```

```
extern int no_coms;                    /* no of comments */
extern int com_sz;                     /* total comment space (in chars) */
extern int var_decls;                        /* no of vars declared */
extern int func_decls;                       /* no of funcs declared */
extern int tl_defs;                    /* total data defs (var + funcs) */
extern int func_defs;                        /* function definitions */
extern int extern_vars;
extern int params;
extern int i_var_decls;
extern int ref_decls;
extern      int e_func_calls;
extern      int operands;


      for (i=0; i < 32; i++) {
            printf("%d ", optrs[i]);
      }
      printf("%d %d %d %d %d %d %d %d %d %d %d %d %d %d %d %d ",
            asm_stats, expr_stats, asign_stats, while_stats, for_stats,
            break_stats, cont_stats, ret_stats, goto_stats, label_stats,
            case_stats, deflt_stats, do_stats, if_stats, switch_stats
            );

      printf("%d %d %d %d %d %d ", var_occ, consts, func_calls,
                                   e_func_calls, no_coms, com_sz
            );

      printf("%d %d %d %d %d %d ", func_defs, ref_decls, extern_vars,
                                   params, i_var_decls, operands
            );

      printf("          ");
}
```

D.3.2 AWK scripts

The following script was used to generate the total size count for each release, using the release_name.size files:

```
BEGIN {
      prev = FILENAME
}

prev != FILENAME {
      print prev, c_files, c_lines, s_files, s_lines

      prev = FILENAME
      c_files = c_lines = s_files = s_lines = 0
}

/.*.c / {      ++c_files;      c_lines += $2   }
/.*.h / {      ++c_files;      c_lines += $2   }
/.*.s / {      ++s_files;      s_lines += $2   }

END {
      print prev, c_files, c_lines, s_files, s_lines
}
```

This script was used to calculate the changes between releases, using the size.data files:

```
# this awk program calulates the number of c-source (*.[ch]) files changed,
# deleted, added and unchanged between two releases
# the two releases must be represented by filelists with
# (compatable) names in the first field and a metric in the second
# it assumes there are no duplicate file entries


BEGIN {
        prev = FILENAME
}
FILENAME != prev {
        file = "second"
}
$1 ~ /.[ch]$/ {
        if (file != "second") {
                size[$1] = $2
        } else {
                if (size[$1] > 0) {
                        if (size[$1] == $2)
                                f_unchanged++
                        else
                                f_changed++
                        size[$1] = "in second"
                } else
                        f_new++
        }
}
END {
        for (i in size) {
                if (size[i] != "in second")
                        f_deleted++
        } # have to do this wasteful processing because this version
          # of awk doesn't have deletion from assosciative arrays
        print prev, FILENAME, f_deleted, f_unchanged, f_changed, f_new
}
```

This script was used to calculate the other metrics (like number of statements, McCabe's and Halstead's metrics) from the release_name.stats files:

```
# this awk program calculates for each release (i.e. each given .stats file):
#       * the number of files for which there are statistics
#       * the number of files for which statistics could not be generated
#       * McCabe's V[G] metric (totalled for the whole release)
#       * Halstead's n1, n2, N1 and N2
#       * the number of function definitions
#       * the number of external links


BEGIN {
       prev_fn = FILENAME
       no_stats = 0
       vg = n1 = n2 = N1 = N2 = n_yes = n_no = n_f_defs = n_ext_lnks = 0
}
prev_fn != FILENAME {
       print prev_fn, n_yes, n_no, no_stats, vg, n1, n2, N1, N2, n_f_defs, n_ext_lnks
       prev_fn = FILENAME
       no_stats = 0
       vg = n1 = n2 = N1 = N2 = n_yes = n_no = n_f_defs = n_ext_lnks = 0
}
NF < 55 {
       ++n_no
}
NF > 55 {
       ++n_yes
       for (i=34; i<=48; i++)
              no_stats += $i
       vg += $37 + $38 + $42 + $44 + $45 + $46 + $47
       N2 += $60
       for (i=2; i<=33; i++) {
              N1 += $i
              if ($i > 0) ++n1
       }
       n2 += $55 + $56 + $58 + $59
       n_f_defs += $55
       n_ext_lnks += $57 + $56
}
END {
       print prev_fn, n_yes, n_no, no_stats, vg, n1, n2, N1, N2, n_f_defs, n_ext_lnks
}
```

ADDENDUM

End of Section 4.3.2 (page 63)

Berkeley data sources

Initially, the BSD/UNIX project was controlled largely by one person (Bill Joy), so the chances of recovering records such as program listings and distribution records depended upon how organized he was. After the restructuring of the BSD/UNIX effort, at the commencement of DARPA sponsorship, much better organized records were kept. Formal release annoucements were made, mechanisms were set up to report bugs and after the release of 4.2, the code was kept under SCCS.

Therefore, it should have been possible to get hold of:

- Distribution tapes and manuals of all released versions of BSD/UNIX
- Release announcements of later versions
- SCCS records of later work
- Distribution records
- Bug report records

End of Section 4.3.3 (page 65)

Berkeley data collection

It turned out that Bill Joy did not keep records from the earliest days. However, CSRG did have most of the recent records of distribution and source changes since 4.2 BSD. Although their archives only went back to 4.1 BSD, 3.0 and 4.0 BSD were obtained from universities which received them. All the distribution tapes had on-line manuals so relatively complete (1 and 2 BSD were not complete UNIX releases) records of BSD/UNIX were available.