IN-61 [illegible]
[illegible]

SOFTWARE ENGINEERING LABORATORY SERIES **SEL-83-003**

[illegible]

COLLECTED SOFTWARE ENGINEERING PAPERS: VOLUME II

NOVEMBER 1983

NASA

National Aeronautics and
Space Administration

Goddard Space Flight Center
Greenbelt, Maryland 20771

SOFTWARE ENGINEERING LABORATORY SERIES SEL-83-003

COLLECTED SOFTWARE ENGINEERING PAPERS: VOLUME II

NOVEMBER 1983

NASA

National Aeronautics and
Space Administration

Goddard Space Flight Center
Greenbelt, Maryland 20771

[illegible] TO
P. 2-3

FOREWORD

The Software Engineering Laboratory (SEL) is an organization sponsored by the National Aeronautics and Space Administration/Goddard Space Flight Center (NASA/GSFC) and created for the purpose of investigating the effectiveness of software engineering technologies when applied to the development of applications software. The SEL was created in 1977 and has three primary organizational members:

NASA/GSFC (Systems Development and Analysis Branch)
The University of Maryland (Computer Sciences Department)
Computer Sciences Corporation (Flight Systems Operation)

The goals of the SEL are (1) to understand the software development process in the GSFC environment; (2) to measure the effect of various methodologies, tools, and models on this process; and (3) to identify and then to apply successful development practices. The activities, findings, and recommendations of the SEL are recorded in the Software Engineering Laboratory Series, a continuing series of reports that includes this document. The papers contained in this document appeared previously as indicated in each section.

Single copies of this document can be obtained by writing to

Frank E. McGarry
Code 582
NASA/GSFC
Greenbelt, Maryland 20771

PRECEDING PAGE BLANK NOT FILMED

PAGE ii INTENTIONALLY BLANK

TABLE OF CONTENTS

Section 1 - Introduction. 1-1

Section 2 - The Software Engineering Laboratory 2-1

"Measuring Software Technology," W. W. Agresti, F. E. McGarry, D. N. Card, et al. . . . . 2-3 -1

"Technical Summary 1982: Report to the National Aeronautics and Space Administration," V. R. Basili 2-9 -2

Section 3 - Resource Models . . . . 3-1 - omit

"Comparison of Regression Modeling Techniques for Resource Estimation," D. N. Card. . . . . 3-3 - 3

"Early Estimation of Resource Expenditures and Program Size," D. N. Card . . . . 3-25 - 4

Section 4 - Software Measures . . . . 4-1 - omit

"Metric Analysis and Data Validation Across FORTRAN Projects," V. R. Basili, R. W. Selby, and T. Phillips . . . . 4-3 -5

"Monitoring Software Development Through Dynamic Variables," C. W. Doerflinger and V. R. Basili. . . 4-45 -6

"Software Errors and Complexity: An Empirical Investigation," V. R. Basili and B. T. Perricone. . 4-73 -7

Section 5 - Data Collection . . . . 5-1 - omit

"A Methodology for Collecting Valid Software Engineering Data," V. R. Basili and D. M. Weiss . . 5-3 7

"Data Collection and Evaluation for Experimental Computer Science Research," M. V. Zelkowitz 5-27

Bibliography of SEL Literature

PRECEDING PAGE BLANK NOT FILMED

PAGE iv INTENTIONALLY BLANK

SECTION 1 – INTRODUCTION

SECTION 1 - INTRODUCTION

This document is a collection of technical papers produced by participants in the Software Engineering Laboratory (SEL) during the period January 1, 1982, through November 30, 1983. The purpose of the document is to make available, in one reference, some results of SEL research that originally appeared in a number of different forums. This is the second such volume of technical papers produced by the SEL. Although these papers cover several topics related to software engineering, they do not encompass the entire scope of SEL activities and interests. Additional information about the SEL and its research efforts may be obtained from the sources listed in the bibliography at the end of this document.

For the convenience of this presentation, the nine papers contained here are grouped into four major categories:

- The Software Engineering Laboratory
- Resource Models
- Software Measures
- Data Collection

The first category presents summaries of the SEL organization, operation, and research activities. The second and third categories include papers describing the results of specific research projects in the areas of resource models and software measures, respectively. The last category presents papers describing strategies for data collection for software engineering research.

The SEL is actively working to increase its understanding and to improve the software development process at Goddard Space Flight Center. Future efforts will be documented in additional volumes of the Collected Software Engineering Papers and other SEL publications.

SECTION 2 – THE SOFTWARE ENGINEERING LABORATORY

SECTION 2 - THE SOFTWARE ENGINEERING LABORATORY

The technical papers included in this section were originally prepared as indicated below.

- Agresti, W. W., F. E. McGarry, D. N. Card, et al., "Measuring Software Technology," Computer Sciences Corporation, Technical Memorandum, November 1983 (reprinted by permission of the authors)

  A version of this paper will appear in Program Transformation and Programmer Environments. New York: Springer-Verlag, 1984.

- Basili, V. R., "Technical Summary - 1982: Report to the National Aeronautics and Space Administration," University of Maryland, Technical Memorandum, December 1982 (reprinted by permission of the author)

D1-61

N87-24896

MEASURING SOFTWARE TECHNOLOGY

W. W. Agresti, D. N. Card, V. E. Church, and G. Page
Computer Sciences Corporation
System Sciences Division
8728 Colesville Road
Silver Spring, Maryland 20910

F. E. McGarry
National Aeronautics and Space Administration
Goddard Space Flight Center
Code 582
Greenbelt, Maryland 20771

ABSTRACT

Results are reported from a series of investigations into the effectiveness of various methods and tools used in a software production environment. The basis for the analysis is a project data base, built through extensive data collection and process instrumentation. The project profiles become an organizational memory, serving as a reference point for an active program of measurement and experimentation on software technology.

INTRODUCTION

Many proposals aimed at improving the software development process have emerged during the past several years. Such approaches as structured design, automated development tools, software metrics, resource estimation models, and special management techniques have been directed at building, maintaining, and estimating the software process and product.

Although the software development community has been presented with these new tools and methods, it is not clear which of them will prove effective in particular environments. When this question is approached from the user's perspective, the issue is to associate with each programming environment a set of enabling conditions and "win" predicates to signal when methods can be applied and which ones will improve performance. Lacking such guidelines, organizations are left to introduce new procedures with little understanding of their likely effect.

Assessing methods and tools for potential application is a central activity of the Software Engineering Laboratory (SEL) [1, 2]. The SEL was established in 1977 by the National Aeronautics and Space

Administration (NASA)/Goddard Space Flight Center (GSFC) in conjunction with Computer Sciences Corporation and the University of Maryland. The SEL's approach is to understand and measure the software development process, measure the effects of new methods through experimentation, and apply those methods and tools that offer improvement. The environment of interest supports flight dynamics applications at NASA/GSFC. This scientific software consists primarily of FORTRAN, with some assembler code, and involves interactive graphics. The average size of a project is 60,000 to 70,000 source lines of code.

SEL investigations demonstrate the advantages of building and maintaining an organizational memory on which to base a program of experimentation and evaluation. Over 40 projects, involving 1.8 million source lines of code, have been monitored since 1977. Project data have been collected from five sources:

- Activity and change forms completed by programmers and managers
- Automated computer accounting information
- Automated tools such as code analyzers
- Subjective evaluations by managers
- Personal interviews

The resulting data base contains over 25 megabytes of profile information on completed projects.

Some highlights of SEL investigations using the project history data base are presented here, organized into three sections:

- Programmer Productivity
- Cost Models
- Technology Evaluations

PROGRAMMER PRODUCTIVITY

The least understood element of the software development process is the behavior of the programmer. One SEL study examined the distribution of programmer time spent on various activities. When specific dates were used to mark the end of one phase and the beginning of the next, 22 percent of the total hours were attributed to the design phase, with 48 percent for coding, and 30 percent for testing. However, if the programmers' completed forms were used to identify actual time spent on various activities, the breakdown was approximately equal for the four categories of designing, coding, testing, and "other" (activities such as travel, training, and unknown) [3]. Although an attractive target for raising productivity was to eliminate the "other" category, the SEL found that this was not easily done.

Regarding individual programmer productivity, the SEL found differences as great as 10 to 1, where productivity was measured in lines of code per unit of effort [4]. This result was consistent with similar studies in other organizations [5].

COST MODELS

Cost is often expressed in terms of the effort required to develop software. In the effort equation,

$$E = aI^b$$

where E equals effort in staff time and I equals size in lines of code, some studies reported a value of b greater than one, indicating that effort must be increased at a higher rate than the increase in system size. The SEL analysis of projects in its data base did not support this result, finding instead a nearly linear relationship between effort and size [6]. This conclusion may be due to the SEL projects being smaller than those that would require more than a linear increase in effort.

In a separate study, the SEL used cost data from projects to evaluate the performance of various resource estimation models. One study, using a subset of completed projects, compared the predictive ability of five models: Doty, SEL, PRICE S, Tecolote, and COCOMO [7]. The objective was to determine which model best characterized the SEL environment. The results showed that some models worked well on some projects, but no model emerged as a single source on which to base a program of estimation [8]. In the SEL environment, cost models have value as a supplementary tool to flag extreme cases and to reinforce the estimates of experienced managers.

TECHNOLOGY EVALUATIONS

Several SEL experiments have been conducted to assess the effectiveness of different process technologies. One study focused on the use of an independent verification and validation (IV&V) team. The premise for introducing an IV&V team into the software development process is that any added cost will be offset by the early discovery of errors. The expected benefit is a software product of greater quality and reliability. In experimenting with an IV&V team in the SEL environment, the benefits were not completely realized [9]. The record on early error detection was better with IV&V than without it, but the reliability of the final product was not improved. Also, the productivity of the development team was comparatively low, due in part to the necessary interaction with the IV&V team. The conclusion was that an IV&V team was not effective in the SEL environment, but may be effective where there are larger projects or higher reliability requirements.

A recent SEL investigation measured the effect of seven specific techniques on productivity and reliability. From the project data base, indices were developed to capture the degree of use of quality assurance procedures, development tools, documentation, structured code, top-down development, code reading, and chief programmer team organization. The results showed that the greatest productivity and reliability improvements due to methodology use lie only in the range of 15 to 30 percent. Significant factors within this range are the positive effect of structured code on productivity and the positive effects of quality assurance, documentation, and code reading on reliability [10].

Figure 1 summarizes the perceived effectiveness of various practices in the the SEL environment [4]. The placement of the models and methods is based on the overhead cost of applying the model or method and the benefit of its use. This summary must be interpreted in the following context:

- The placement reflects subjective evaluations as well as experimental results.
- The chart is indicative of experiences in the SEL environment only.
- The dynamic nature of the situation is not apparent. The evaluation may reflect on an earlier and less effective example of the model or method.

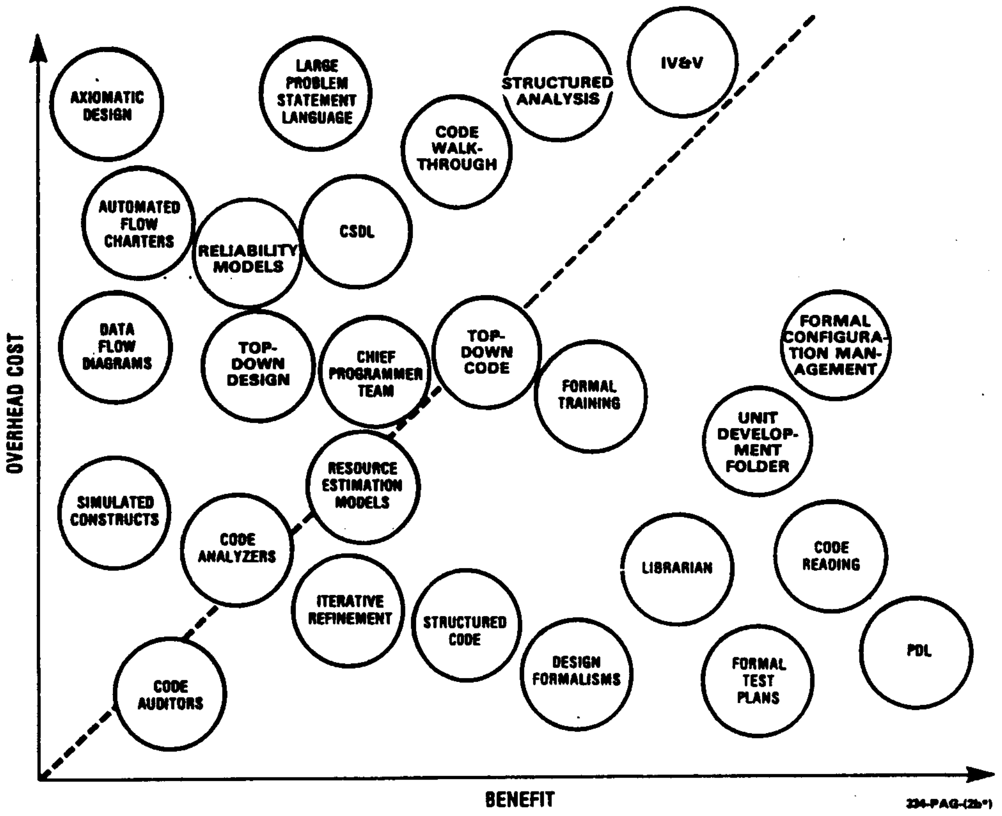


Figure 1. What Has Been Successful in Our Environment?

CONCLUSIONS

The experiences of the SEL demonstrate that statistically valid evaluation is possible in the software development environment, but only if the prerequisite quantitative characterization of the process has been obtained. Through its program of assessing and applying new methods and tools, the SEL is actively pursuing the creation of a more productive software development environment.

REFERENCES

1. V. R. Basili and M. V. Zelkowitz, "Operation of the Software Engineering Laboratory," Proceedings of the Second U.S. Army/IEEE Software Life Cycle Management Workshop. New York: Computer Societies Press, 1978

2. D. N. Card, F. E. McGarry, G. Page, et al., SEL-81-104, The Software Engineering Laboratory, Software Engineering Laboratory, 1982

3. E. Chen and M. V. Zelkowitz, "Use of Cluster Analysis To Evaluate Software Engineering Methodologies," Proceedings of the Fifth International Conference on Software Engineering. New York: Computer Societies Press, 1981

4. F. E. McGarry, "What Have We Learned in the Last Six Years Measuring Software Development Technology," SEL-82-007, Proceedings of the Seventh Annual Software Engineering Workshop, Software Engineering Laboratory, 1982

5. H. Sackman, W. J. Erikson, and E. E. Grant, "Exploratory Experimental Studies Comparing Online and Offline Program Performance," Communications of the ACM, January 1968, vol. 11, no.1, pp. 3-11

6. J. W. Bailey and V. R. Basili, "A Meta-Model for Software Development Resource Expenditures," Proceedings of the Fifth International Conference on Software Engineering. New York: Computer Societies Press, 1981

7. IIT Research Institute, Quantitative Software Models, Rome Air Development Center, New York, 1979

8. J. Cook and F. E. McGarry, SEL-80-007, An Appraisal of Selected Cost/Resource Estimation Models for Software Systems, Software Engineering Laboratory, 1980

9. G. Page, "Methodology Evaluation: Effects of Independent Verification and Integration on One Class of Application," SEL-81-013, Proceedings of the Sixth Annual Software Engineering Workshop, Software Engineering Laboratory, 1981

10. D. N. Card, F. E. McGarry, and G. Page, "Evaluating Software Engineering Methodologies in the SEL" (paper presented at Sixth Minnowbrook Workshop on Software Performance Evaluation, Minnowbrook, New York, 1983)

D2-61

N87-24897

TECHNICAL SUMMARY
1982

REPORT TO THE NATIONAL AERONAUTICS
AND SPACE ADMINISTRATION

Grant 01-526104

Department of Computer Science
University of Maryland
College Park, MD 20742

Principal Investigator:
Dr. Victor Basili

Overview

During 1982, in conjunction with NASA/GSFC Software Engineering Laboratory (SEL), research was conducted in 4 areas: Software Development Predictors, Error Analysis, Reliability Models and Software Metric Analysis. Summaries of the projects follow below.

1. Software Development Predictors

A study is being done on the use of dynamic characteristics as predictors for software development. It is hoped that by examining a set of readily available characteristics, the project manager may be able to determine such things as when a project is in trouble and evaluate the quality of the product as it is being designed.

Project DEB was selected as the control for the project since it was considered fairly successful and is well documented. Information found in the history files and resource summary files was initially utilized. These files were chosen because the information they contain is readily accessible to the manager (ie. number of lines of code, manpower, computer time, etc.). Several profiles of project DEB were then made using this information. Project DEA's profiles were then compared with these results. This project was chosen because it was very similar to DEB but was considered less successful.

The history file was first examined to see if any growth pattern existed for the lines of code. The initial look at DEA and DEB looked hopeful but further investigation of other projects showed no discernible pattern. Other examinations of this file yielded similar results.

When a comparison of the information in the history and resource summary files was made some differences did appear. Initial plots used accumulative totals versus different time factors. These plots did demonstrate visible differences between the two projects. Further investigation using weekly totals instead of accumulative totals showed an even larger difference between the projects.

Project DEA had a higher frequency of changes at the beginning of the project, while at the same time, the number of hours of manpower reported for the interval was less. The number of computer runs made was higher for DEB in the part of the project where DEA was experiencing the higher number of changes per manpower. In all, project DEA appears to have had less effort placed during the early phase of the project which may of led to the problems in the end. Another important aspect of project DEA was that several thousand lines of code appear to have been transported. Adaptation of this code may explain the high number of changes initially seen in DEA.

From this examination the following general goals and hypothesis have been generated:

A) The manpower usage in the SEL environment is a discernible pattern and may be used as a predictor.

1) The ideal staffing for a successful project is a two hump curve with the second hump beginning roughly 2/3 into the project.

2) The two humps mentioned in hypothesis 1 should peak at approximately the same height.

3) The maximum peak height of the first hump is proportional to the final size of the project. This also hold for the second hump based on hypothesis two.

4) The location of the two peaks is constant with relation to the amount of manpower utilized.

5) The amount of manpower expended between the two peaks is constant.

6) Projects deemed less successful by subjective analysis have sharp changes in the amount of manpower spent per change.

B) The pattern of changes in relation to manpower, computer runs, lines of code, etc. may be used as a predictor in the SEL environment.

1) The amount of manpower to make a change should increase toward the end of a project and be stable at the beginning.

2) The manpower per change should be lower in the beginning of the project. See also goal D.

3) Projects deemed less successful by subjective analysis have sharp changes in the amount of manpower spent per change.

4) The ratio of changes to computer run should decrease as the project evolves.

5) The amount of computer time spent on detecting and correcting a given change will remain constant.

C) The number of computer runs is closely related to the development of a project and may be used to judge project development.

1) The number of computer runs remains constant during the initial hump of the staffing curve. The number of computer runs will drop during the second hump of the staffing curve.

2) The ratio of changes to computer runs should decrease as the project evolves.

D) A close examination of the types of changes and the pattern they make over time should be a good indication of the success of a given project.

1) Time consuming changes that occur late in the project more often appear in modified code.

2) Unit testing is not as extensive on modules with modified code. Undetected errors may cause major problems latter in development.

3) The types of changes vary across the development of a project.

4) The number of changes per hour of manpower is related to the type of changes being done.

5) The types of change that require more time to correct occur during the second staffing hump.

Several projects will now examined to test the validity of these finds. The change report forms will also be examined to see if the information in them yields any useful predictors.

To conclude, the study has completed its initial analysis of the two projects. It appears there are some significant factors that could be useful as predictors. Further analysis may yield some information that would be useful to a project manager.

2. Error Analysis

A). Publication of existing results -- Three papers are being prepared from earlier work on error analysis conducted by the SEL laboratory. One is on the data collection methodology and the validation of the accuracy of the data, the second one is on the analysis of the SEL projects directly and the third one is a comparison of the SEL projects with projects of the Naval Research Laboratory. These papers are currently being submitted for publication and will be published as University of Maryland Technical Reports in the interim.

B). A study on software errors and complexity -- The distribution and relationships derived from the change data collected during the development of the medium scale satellite project shows that meaningful results can be obtained which allow insight into software traits and the environment in which it is developed. The project studied in this case was GMAS. Modified and new modules were shown to behave similarly. An abstract classification scheme for errors which allows a better understanding of the overall traits of a software project was also provided. Finally, various size and complexity metrics are examined with respect to errors detected within the software yielding some interesting results. A University of Maryland Technical Report describing these results was published [Bas82]. This paper has been submitted for publication.

C). A further examination of the error characteristics of the DE_A and DE_B projects is currently being undertaken. This error analysis is being conducted using the techniques developed and documented in [Wei81] and [Per82]. The focal point of this research effort is to characterize errors in the NASA/GSFC software development environment.

A preliminary review of a sample of the Change Report Forms from both DE_A and DE_B has been conducted. The sample included only those CRF's for which an error change was reported. The purpose of this review was to 'get a flavor' for the data collected and to preliminarily assess the consistency of that data with the results found to date by SEL personnel.

The sample included 98 CRF's from DE_A and 90 CRF's from DE_B. Of the 98 CRF's from DE_A, 63 (64.3%) of the errors were classified as an 'error in the design or implementation of a single component.' Of the 90 CRF's from DE_B, 16 errors were reported as 'clerical errors.' Of the remaining 74 DE_B errors (non-clerical errors), 61 (84.2%) of the errors were also classified as 'errors in the design or implementation of a single component.'

Although the percentage classified as 'errors in a single component' for DE_B was higher than the other studies, these preliminary results appear to follow the results of previous analyses [Wei81]. As in that previous work, the distribution of errors in other categories does not neatly fit a pattern. In fact, there are too few events in the other categories to draw any initial conclusions. It will be interesting to explore the reason(s) DE_B experienced a substantially larger number of 'clerical errors.'

There are marked differences in the remaining DE_A and DE_B error reports. This may be attributable to the reported differences in the two projects. It is not possible at this time to conjecture on more tangible causes for the differences. The full set of error change reports will have to be examined, for both projects.

It is worth noting here that for DE_A, 31 of 98 error reports (31.6%) examined were classified as being an ′error in the design or implementation of more than one component.′ Based on previous results cited above, this is an unusually high percentage. Only 4 components (4.1%) had errors reported that were not in the design or implementation of component(s) categories.

As part of the preliminary work toward the above goal, the related literature released by SEL was reviewed. A conclusion reached was that the definitions of several critical terms were not necessarily consistent, and often times the technical reports make too great an assumption about the uniformity of use of software engineering terms.

′Interface′ provides a good example of an ill-defined yet oft used term. Using the definition from [Wei81] (the same definition is used in [Bas80b] and [Glo79]) it is arguable that interface errors can be captured five ways from the CRF:

-an error involving more than one component;

-an error involving a common routine;

-from textual comments in the CRF (eg: a CRF for which the error was entered as having affected one component but the text indicated that the error was in a subroutine call statement);

-an error reported as having been located in one component but the change required to repair the error affected more than one component; and

-a change that caused an error because either the change invalidated an assumption made elsewhere in the software or an assumption made about the rest of the software in the design of the change was incorrect (contingent on ability to capture supporting text and ability to distinguish from erroneous assumptions made about a single component).

An effort is currently underway to develop a more restrictive set of definitions for software engineering terms, specifically those that apply to error analysis. The basis of this effort is the set of definitions published in [Bas80] and [Glo79] and will be modified, as necessary, in consultation with those persons associated with SEL in the past and present, whose work is or was related to the error analysis effort.

3. Reliability Models

A study is being performed in the area of reliability models. This research includes the field of program testing because the validity of some reliability models depends on the answers to some unanswered questions about testing.

The eventual goal of this research is to understand how and when to use reliability models. We are investigating the use of functional testing because some reliability models make assumptions about the way program testing is accomplished [Musa]. It is not known if functional testing satisfies the random testing assumptions made by the reliability models. The validity of reliability models that use data generated by functional testing is uncertain until this question is answered.

We are using structural coverage metrics to gain further insight into the effects of functional testing. A structural coverage metric is a measure of how much of a program was executed for given input data. Studying the coverage metric may allow us to develop other measures of reliability.

An additional bonus of this research is that it allows us to compare functional testing and structural testing. It is not known how these two methods of testing are related. The results of this investigation may answer that question.

Since January background material has been studied with regard to reliability models, and functional and structural testing [Mueller]. A FORTRAN preprocessor has been written to calculate the structural coverage metrics of GSFC FORTRAN source code.

The preprocessor calculates the simplest metric, the percent of executable code that is executed. There are several ways to measure coverage [Auerbach]. One method uses interpretation of the source code. The interpreter records which statements are executed. At the end of interpretation, it writes a list of executed statements.

The second method uses "switches", small sections of code that are inserted into the source program text wherever the flow of control diverges or converges. The switch has 2 values: 0 if it was not executed, 1 if it was executed. The value of the switches is output after execution.

An example:

```
INTEGER SWITCH ( N )

FOR I = 1, N
SWITCH (I) = 0
        .
        .
        .
READ ( J );
IF ( even ( J ))
        THEN
                SWITCH ( 1 ) = 1;
                .
                .
                .
        ELSE
                SWITCH ( 2 ) = 1;
                .
```

```
                    .
                    .
        ENDIF
        .
        .
        .
FOR I = 1, N
        WRITE ( SWITCH ( I ));
END
```

When this program is executed, one of the two branches of the if statement will be executed. By examining the values of the array SWITCH, we can determine what code was executed. By analyzing the code and counting statements, the number of statements executed can be determined. In practice, the amount of data generated will be large. Software tools are needed to help analyze the data.

The switches can be inserted by a preprocessor (before compilation) or by a compiler (during compilation). The switches may be in-line code (as in the example) or a call to a switch subroutine that records the flow of control.

This latter approach was taken and a preprocessor was developed that runs on VAX/Unix at UMCP. The preprocessor takes a copy of the input source code, and modifies it. This modified copy will be returned to the source computer (at GSFC) where it will be compiled and executed. The execution produces the desired coverage data. The coverage data will be returned to the University for analysis.

Many things remain to be done before we reach our goal of understanding how and when to use reliability models. The immediate goal is to try to answer the functional testing / reliability model question. The project RADMAS has been chosen as an experimental system [CSC]. The preprocessor must be used to modify the RADMAS source code. (The RADMAS project and its functionally-generated acceptance tests have been made available for the coverage experiment.) The modified RADMAS code must be executed at GSFC using the functionally-generated acceptance tests.

This experiment should answer these questions about functional testing and reliability models:

-What is the percent coverage of functional testing?

-Does functional testing meet the randomness requirements of the MTTF models? If not, can it be made to?

-Do the structural metrics show any useful patterns in the way that functional testing tests programs? How does the coverage set grow? At what rate does the coverage set grow?

-How independent are individual tests from a coverage point of view?

The results of this experiment will raise further questions about functional testing and reliability models. This will require more experimentation. If these questions are answered, there is more work to do concerning how and when to use reliability models.

4. Software Metrics.

The attraction of the ability to predict the effort in developing or explain the quality of software has led to the proposal of several theories and metrics [Hal77, McC76, Gaf, Che78, Cur79]. In the Software Engineering Laboratory, the Halstead metrics, McCabe's cyclomatic complexity and various standard metrics have been analyzed for their relation to effort, development errors and one another [Bas82a]. This study examined data collected from seven SEL (FORTRAN) projects and applied three effort reporting accuracy checks to demonstrate the need to validate a database.

The investigation examined the correlations of the various metrics with effort (functional specifications through acceptance testing) and development errors (both discrete and weighted according to amount of time to locate and fix) across several projects at once, within individual projects and for individual programmers across projects.

In order to remove the dependency of the distribution of the correlation coefficients on the actual measures of effort and errors, the non-parametric Spearman rank-order correlation coefficients were examined [Ken79]. The metrics' correlations with actual effort seem to be strongest when modules developed entirely by individual programmers or taken from certain validated projects are considered. When examining modules developed totally by individual programmers, two averages formed from the proposed validity ratios induce a statistically significant ordering of the magnitude of several of the metrics' correlations. The systematic application of one of the data reliability checks (the frequency of effort reporting) substantially improves either all or several of the projects' effort correlations with the metrics. In addition to these relationships, the Halstead metrics seem to possess reasonable correspondence with their estimators, although some of them have size dependent properties. In comparing the strongest correlations, neither Halstead's E metric, McCabes' cyclomatic complexity nor source lines of code relates convincingly better with effort than the others.

The metrics examined in this study were calculated from primitive measures derived from a source analyzing program (SAP -- Revision I) [Dec82]. An earlier version of this static analyzer implemented a less comprehensive definition of Halstead operators and operands[O'Ne78].

Some work has been done comparing the metrics´ correlations when they have been determined from the different interpretations of the primitive measures.

This investigation has been submitted for publication to the Transactions on Software Engineering and will appear as a University of Maryland Technical Report.

5. References

[Auerbach] Auerbach Publishers Inc., "Practical Measures for Program Testing Thoroughness", 1977.

[Bas80] V. Basili, Tutorial on Models and Metrics for Software Management and Engineering, p. 340, IEEE 1980

[Bas82a] V. Basili, R. Selby and T. Phillips, "Data Validation in a Software Metric Analysis of FORTRAN Modules," -- to appear IEEE Transactions on Software Engineering, July 1982.

[Bas82b] V. Basili and B. Perricone, "Software Errors and Complexity: An Empirical Investigation," The Software Engineering Laboratory, University of Maryland Technical Report TR-1195, August 1982

[Bas82c] V. Basili, "An Assessment of Software Measures in the Software Engineering Laboratory," presented at Goddard Space Flight Center, January 1982.

[Card82] Card, D., F.McGarry and J. Page, "Evaluation of Management Measures of Software Development," Vol I & II, Software Engineering Laboratory Series, SEL - 82 - 001, Goddard Space Flight Center, September 1982.

[Chen 78] E. T. Chen, "Program Complexity and Programmer Productivity," IEEE Transactions on Software Engineering, Vol. SE-4, No. 3, pp. 187-194 (May 1978).

[CSC] Computer Sciences Corporation, RADMAS User's Guide., September 1981.

[Curtis et al 79] Curtis, Sheppard and Milliman, "Third Time Charm: Stronger Prediction of Programmer Performance by Software Complexity Metrics," Proceedings of the Fourth International on Software Engineering, pp. 356-360 (1979).

[Decker & Taylor 82] W. J Decker and W. A. Taylor, "FORTRAN Static Source Code Analyzer Program (SAP) User's Guide (Revision 1)," SEL-78-102, Software Engineering Laboratory, (May 1982).

[Gaffney & Heller] J. Gaffney and G. L. Heller, "Macro Variable Software Models for Application to Improved Software Development Management," Proceedings of Workshop on Quantitative Software Models for Reliability, Complexity and Cost, IEEE Computer Society.

[Glo79] S. Gloss-Soler, The DACS Glossary -- A Bibliography of Software Engineering Terms, Data and Analysis Center for Software, p. 56, October 1979

[Halstead 77] M. Halstead, Elements of Software Science, Elsevier North-Holland, New York (1977).

[Kendall & Stuart 79] M. Kendall and A. Stuart, The Advanced Theory of Statistics, Vol. 2, Fourth Ed., MacMillian, New York, 1979, pp. 503-508.

[McCabe 76] T. J. McCabe, "A Complexity Measure," IEEE Transactions on Software Engineering, Vol. SE-2, pp. 308-320 (December 1976).

[Mueller] Mueller, Barbara, "Test Data Selection: A Comparison of Structural and Functional Testing", April 1980, private paper.

[Musa] Musa, John, D., "Software Reliability Management", Software Life Cycle Management Workshop, August 1977.

[O'Neill et al 78] E. M. O'Neill, S. R. Waligora and C. E. Goorevich, "FORTRAN Static Source Code Analyzer (SAP) User's Guide," SEL-78-002, Software Engineering Laboratory (February 1978).

[Pic82] G Picasso, "The Rayleigh Curve as a Model for Effort Distribution Over the Life of Medium Scale Software Systems," Department of Computer Science, University of Maryland Technical Report TR-1186, July 1982.

[Wei81] D. Weiss, "Evaluating Software Development by Analysis of Change Data," The Software Engineering Laboratory, University of Maryland Technical Report TR-1120, November 1981

SECTION 3 – RESOURCE MODELS

omit

SECTION 3 - RESOURCE MODELS

The technical papers included in this section were originally prepared as indicated below.

- Card, D. N., "Comparison of Regression Modeling Techniques for Resource Estimation," Computer Sciences Corporation, Technical Memorandum, November 1982 (reprinted by permission of the author)
- Card, D. N., "Early Estimation of Resource Expenditures and Program Size," Computer Sciences Corporation, Technical Memorandum, June 1982 (reprinted by permission of the author)

D3-61

N87-24898

COMPARISON OF REGRESSION MODELING TECHNIQUES FOR RESOURCE ESTIMATION

Prepared by

COMPUTER SCIENCES CORPORATION

D. N. Card

For

GODDARD SPACE FLIGHT CENTER

Under

Contract NAS 5-27555

Task Assignment 85100

November 1982

INTRODUCTION

The development and validation of resource utilization models has been an active area of software engineering research. Regression analysis is the principal tool employed in these studies. However, little attention has been given to determining which of the various regression methods available is the most appropriate. The objective of the study presented in this memorandum is to compare three alternative regression procedures by examining the results of their application to one commonly accepted equation for resource estimation. This memorandum summarizes the data studied, describes the resource estimation equation, explains the regression procedures, and compares the results obtained from the procedures.

DATA SUMMARY

This study is based on data collected from 22 flight dynamics software projects studied by the Software Engineering Laboratory (SEL). The general class of flight dynamics software includes applications to support attitude determination, attitude control, maneuver planning, orbit adjustment, and mission analysis (Reference 1). The specific projects selected for this analysis were developed in FORTRAN for operation on the same computer system. The range of system size (developed lines of source code) and development effort (staff-months) for these 22 projects is indicated in Table 1.

THE RESOURCE ESTIMATION EQUATION

Variations of one basic equation have been incorporated in many resource estimation models (Reference 2). This equation relates project size to development effort. Additive and/or multiplicative factors based on experience, complexity, software tape, etc. are added to form more sensitive models. The SEL also has developed a model based on this equation (Reference 3). The general form of the estimating equation is

$$H = AL^B \qquad (1)$$

where

H = staff-months of effort
L = lines of source code
A, B are constants

Table 1. Summary of Measures

MEASURE	MEAN	STANDARD DEVIATION	MINIMUM	MAXIMUM
HP[1]	37.07	32.23	2.99	93.85
HPM[2]	47.30	40.73	4.31	116.28
HPMO[3]	54.80	47.42	4.96	138.32
DEV-PROD[4]	34.20	31.48	1.95	104.53
LOG (HP)	3.11	1.13	1.10	4.54
LOG (HPM)	3.35	1.13	1.46	4.76
LOG (HPMO)	3.50	1.13	1.60	4.93
LOG (DEV-PROD)	+2.99	1.18	.67	4.65

[1] - Programmer staff-month
[2] - Programmer and Manager Staff-months
[3] - Programmer, Manager, and Other Staff-months
[4] - Developed Lines of Source Code

Because the projects studied by the SEL include a substantial proportion of reused code, a "developed" lines of source code measure was devised to account for the higher productivity due to reusing code (Reference 3). The equation for computing developed lines of source code is

$$L = N + E + 0.2S + 0.2U \qquad (2)$$

Where

L = developed lines of source code
N = newly coded lines
E = extensively modified lines
S = slightly modified lines
U = lines reused unchanged

This software product measure (L) can be related to three measures of development effort (H). These measures, as they are defined for the subsequent analysis, are the following:

- HP - programmer staff-months of effort
- HPM - programmer and manager staff-months of effort
- HPMO - programmer, manager, and other (total) staff-months of effort

ALTERNATIVE REGRESSION PROCEDURES

Three alternative regression procedures are availabe for deriving values for the constants in Equation 1. These are the following:

- Non-linear regression of original data
- Linear regression of original data
- Linear regression of logarithmically transformed data

A non-linear regression procedure can find a least-squares solution for the constants in Equation 1 without requiring either a manipulation of the equation or a transformation of the data. Several such algorithms have been implemented. However, the calculation of non-linear solutions is computationally intensive. Thus, it consumes a substantial amount of computer resources. Reference 4 describes the derivative-free algorithm used in this study.

Equation 1 can be reduced to a linear form by fixing the value of the exponent (B) at 1.0. The resulting equation is the following:

$$H = AL \qquad (3)$$

Then ordinary linear least-squares regression can be applied to the untransformed data. Unfortunately, this simple solution ignores the conceptual importance of a potential exponential relationship between software size and development effort.

This relationship can be captured by performing a logarithmic transformation of Equation 1 and the data. The resulting equation is

$$\text{Log}(H) = \text{Log}(A) + B\,\text{Log}(L) \qquad (4)$$

Solutions for A and B in this equation can be derived by ordinary linear least-squares regression. Although this procedure is computationally less intensive than the non-linear procedure, it requires a prior transformation of the data. The range of the logarithmically transformed data is shown in Table 1.

COMPARISON OF RESULTING MODELS

Each of the regression procedures described in the previous section were applied to the data for each measure of effort. These analyses were performed with the Statistical Analysis System software package (Reference 5). Table 2 summarizes the results. The goodness-of-fit obtained by any regression model is measured by the mean square error (MSE) and correlation coefficient (R). Unfortunately, as shown in Table 2, these values are not directly comparable for all the regression models considered here.

However, it is clear from Table 2 that for all measures of effort the results provided by the linear and log-linear procedures are very similar. The estimates of A and B for the log-linear model (Equation 4) are close to those of the linear model (Equation 3); slight decreases in B in the log-linear case are compensated by increases in A. Furthermore, the correlation coefficients obtained by the two procedures are nearly identical in all three cases. Therefore, the linear regression procedure produces a model as good as that of the log-linear procedure in a considerably more straightforward manner.

The model produced by the non-linear procedure differs considerably from those produced by the linear and log-linear procedures (see Table 2). The values of B (Equation 1) depart significantly from 1.0; the relationship defined is clearly exponential. Furthermore, the mean square error of the non-linear model is substantially less than that of the linear model. Although a direct comparison between the non-linear and log-linear models (in terms of MSE or R) is not possible, the log-linear model is so close to the linear model that we can safely conclude that the non-linear model is the most accurate of the three.

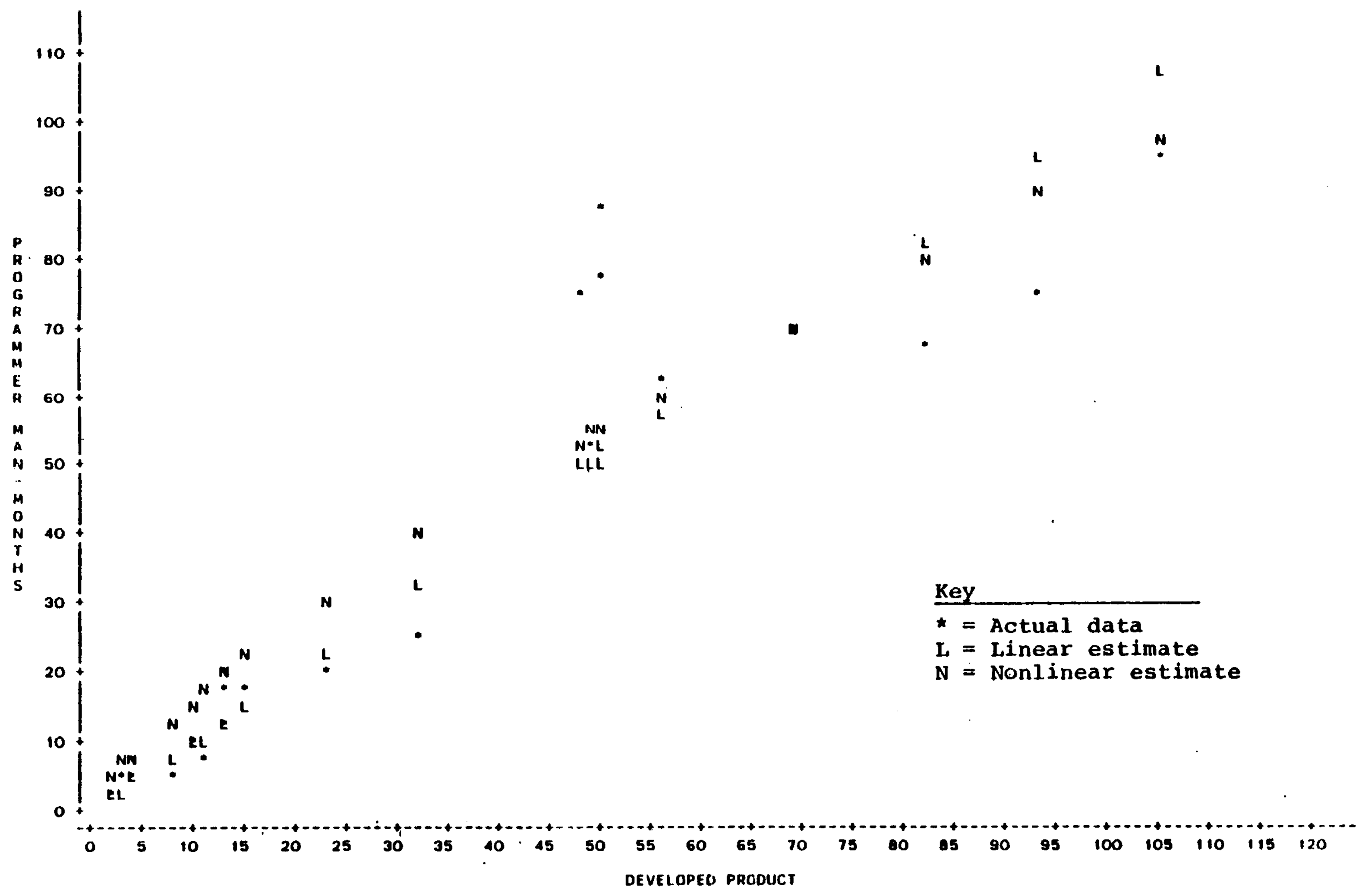
Figures 1 through 3 illustrate the relationships between system size and development effort defined by the linear and non-linear models. (The log-linear model is not shown because it is so similar to the linear model). A cursory examination of these figures indicates that the linear model fits the data at the low end of the range better while the non-linear model fits the data at the high end of the range better.

Table 2. Comparison of Model Results[a]

MEASURE OF EFFORT	MODEL[b]	ESTIMATES A	ESTIMATES B	MEAN SQUARE ERROR	CORRELATION COEFFICIENT	DETAILED TABLE
HP	Nonlinear	2.74	.77	146	-[c]	A-1
	Linear	1.02	1.00[d]	175	.93	A-2
	Loglinear	1.40	.93	-[e]	.94	A-3
HPM	Nonlinear	3.86	.74	258	-[c]	A-4
	Linear	1.23	1.00[d]	321	.92	A-5
	Loglinear	1.81	.92	-[e]	.93	A-6
HPMO	Nonlinear	4.03	.77	306	-[c]	A-7
	Linear	1.51	1.00[d]	370	.93	A-8
	Loglinear	2.11	.92	-[e]	.93	A-9

a - Detailed tables are contained in the Appendix
b - Nonlinear: $H \doteq AL^B$
Linear: $H = AL$
Log-Linear: $\text{Log } H = \text{Log } A + B \cdot \text{Log } L$
c - Not applicable
d - Fixed value
e - Magnitude not comparable

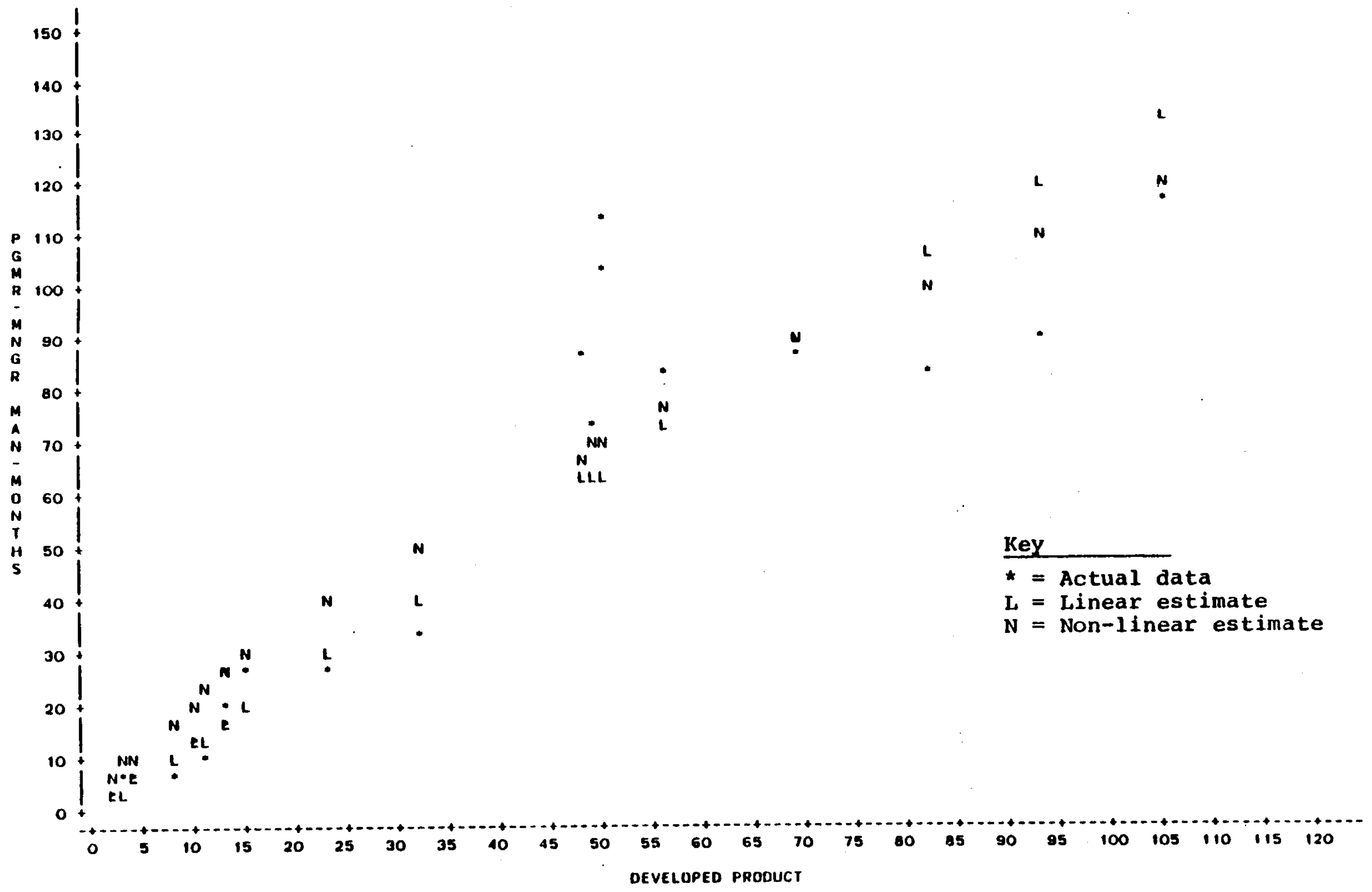
5



NOTE: 7 OBS HIDDEN

Figure 1. Comparison of Models for Programmer Staff-Months

10



NOTE: 9 OBS HIDDEN

Figure 2. Comparison of Models for Programmer and Manager Staff Hours

15

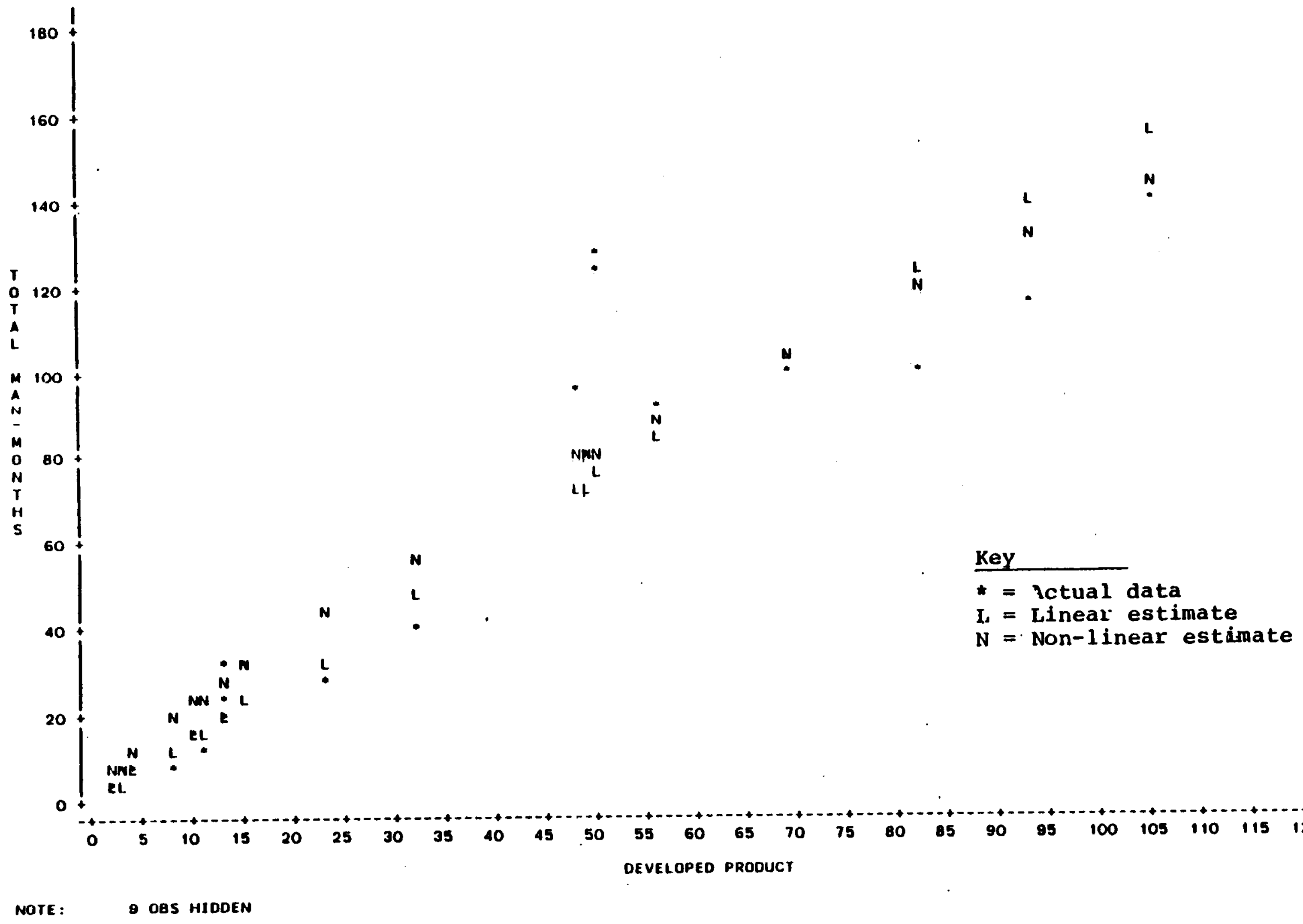


Figure 3. Comparison of Models for Total Staff-Months

This phenomenon suggests an explanation for the closeness of the log-linear model to the linear model. The effect of the logarithmic transformation is to weight smaller data values relatively higher; Table 1 shows that large data values are affected more dramatically by the transformation. Thus, the log-linear regression procedure produces a nearly linear result because it is weighted in favor of smaller data values where observation indicates that the relationship between system size and development effort is most nearly linear.

CONCLUSION

The non-linear regression procedure emerges from this study as the superior technique. The foregoing evaluation of the three alternative regression procedures is summarized in Table 3. The total rating of each procedure shown in the table would be changed if the three elements, of which it is composed (numerical accuracy, conceptual accuracy, and computational cost), were not weighted equally.

In addition to the implication for the choice of statistical techniques, the results of the study suggest some other factors that should be considered in future research. The estimate of the exponent (B) derived by each procedure is fairly constant for all measures of effort (see Table 2). The additional effort contributed by managers and others is accounted for by an increase in the multiplicative factor (A) for the HPM and HPMC measures of effort. Furthermore, the effort contributed by managers and other nonprogrammer personnel is strongly affected by the complexity of a project, the experience of the development team, and the development methodologies employed. This confirms that these other effects should be represented as multiplicative factors in a comprehensive resource estimation model. Published models generally have taken this approach.

The exponential relationship, illustrated in Figures 1 through 3 has another implication. Although the relationship between system size and development effort is nearly linear for small systems, the development effort <u>due to size alone</u> does not increase in proportion to size for <u>large systems</u>. This suggests that the influences of factors such as methodology, experience, and complexity, may be more important for large systems.

The results of this study allow the optimistic conclusion that the basic relationship presented in Equation 1 provides a sufficient framework for the construction of comprehensive resource estimation models when the appropriate statistical techniques are applied.

Table 3. Relative Ratings of Regression Procedures[a]

REGRESSION PROCEDURE	NUMERICAL ACCURACY	CONCEPTUAL ACCURACY	COMPUTATIONAL COST	TOTAL RATING
Non-linear	1	1.5	3	5.5
Linear	2.5	3	1	6.5
Log-Linear	2.5	1.5	2	6.0

[a]Lowest rating is highest accuracy, lowest cost. Average ranks are awarded in cases of tied ratings.

APPENDIX - REGRESSION ANALYSIS RESULTS

This appendix reproduces the computer generated output from which Table 2 was compiled. The following detailed tables are included:

Table	Content
A-1	Non-Linear Model for Programmer Staff-Months
A-2	Linear Model for Programmer Staff-Months
A-3	Log-Linear for Programmer Staff-Months
A-4	Non-Linear Model for Programmer and Manager Staff-Months
A-5	Linear Model for Programmer and Manager Staff-Months
A-6	Log-Linear Model for Programmer and Manager Staff-Months
A-7	Non-Linear Model for Total Staff-Months
A-8	Linear Model for Total Staff-Months
A-9	Log-Linear Model for Total Staff-Months

Table A-1. Non-Linear Model for Programmer Staff-Months

```
          NON-LINEAR LEAST SQUARES SUMMARY STATISTICS     DEPENDENT VARIABLE HP

             SOURCE                DF       SUM OF SQUARES          MEAN SQUARE

             REGRESSION             2       49130.68883215       24565.34441607
             RESIDUAL              20        2916.83847375         145.84192369
             UNCORRECTED TOTAL     22       52047.52730590

             (CORRECTED TOTAL)     21       21820.84200998

PARAMETER           ESTIMATE            ASYMPTOTIC                ASYMPTOTIC 95 %
                                        STD. ERROR               CONFIDENCE INTERVAL
                                                              LOWER              UPPER
A                 2.74366755            1.14789137         0.34922616         5.13810894
B                 0.76657413            0.09981586         0.55836346         0.97478480

                      ASYMPTOTIC CORRELATION MATRIX OF THE PARAMETERS

                                                A         B

                                   A        1.000000 -0.991484
                                   B       -0.991484  1.000000

NOTE: ALL ASYMPTOTIC STATISTICS ARE APPROXIMATE. REFERENCE: RALSTON AND JENNRICH, TECHNOMETRICS, FEBRUARY 1978, P 7-14.
```

Table A-2. Linear Model for Programmer Staff-Months

```
                                           GENERAL LINEAR MODELS PROCEDURE

DEPENDENT VARIABLE: HP        PROGRAMMER MAN-MONTHS

SOURCE                DF      SUM OF SQUARES        MEAN SQUARE      F VALUE          PR > F       R-SQUARE           C.V.

MODEL                  1      48365.34712971     48365.34712971       275.83          0.0001       0.929254        35.7239

ERROR                 21       3682.18017619       175.34191315                      STD DEV                       HP MEAN

UNCORRECTED TOTAL     22      52047.52730590                                     13.24167335                   37.06669989

SOURCE                DF           TYPE I SS       F VALUE       PR > F        DF        TYPE IV SS      F VALUE      PR > F

DEV_PROD               1      48365.34712971        275.83       0.0001         1    48365.34712971       275.83      0.0001

                                   T FOR HO:     PR > |T|           STD ERROR OF
PARAMETER              ESTIMATE   PARAMETER=0                          ESTIMATE

DEV_PROD             1.01927651         16.61       0.0001            0.06137164
```

Table A-3. Log-Linear Model for Programmer Staff-Months

```
                                        GENERAL LINEAR MODELS PROCEDURE

DEPENDENT VARIABLE: HPT

SOURCE                  DF      SUM OF SQUARES       MEAN SQUARE       F VALUE         PR > F      R-SQUARE          C.V.

MODEL                    1         25.11535373       25.11535373        321.04         0.0001      0.941356        9.0034

ERROR                   20          1.55461205        0.07823060                      STD DEV                   HPT MEAN

CORRECTED TOTAL         21         26.67996578                                     0.27969734                 3.10657908


SOURCE                  DF           TYPE I SS      F VALUE     PR > F       DF      TYPE IV SS      F VALUE       PR > F

DEVPRODT                 1         25.11535373       321.04     0.0001        1     25.11535373       321.04       0.0001


                                   T FOR HO:      PR > |T|          STD ERROR OF
PARAMETER             ESTIMATE    PARAMETER=0                         ESTIMATE

INTERCEPT           0.33622917           2.03       0.0560           0.16571639
DEVPRODT            0.92572772          17.92       0.0001           0.05166566
```

Table A-4. Non-Linear Model for Programmer and Manager Staff-Months

```
          NON-LINEAR LEAST SQUARES SUMMARY STATISTICS     DEPENDENT VARIABLE HPM

             SOURCE                DF       SUM OF SQUARES        MEAN SQUARE

             REGRESSION             2       78885.66694100     39442.83347050
             RESIDUAL              20        5167.54816982       258.37740949
             UNCORRECTED TOTAL     22       84053.21513081

             (CORRECTED TOTAL)     21       34830.14939308

 PARAMETER            ESTIMATE          ASYMPTOTIC               ASYMPTOTIC 95 %
                                        STD. ERROR             CONFIDENCE INTERVAL
                                                              LOWER            UPPER
 A                   3.86262913         1.64840504         0.42414239       7.30111588
 B                   0.74114923         0.10228988         0.52777790       0.95452057

                 ASYMPTOTIC CORRELATION MATRIX OF THE PARAMETERS

                                          A          B

                       A           1.000000  -0.991088
                       B          -0.991088   1.000000

NOTE: ALL ASYMPTOTIC STATISTICS ARE APPROXIMATE. REFERENCE: RALSTON AND JENNRICH, TECHNOMETRICS, FEBRUARY 1978, P 7-14.
```

Table A-5. Linear Model for Programmer and Manager Staff-Months

```
                                        GENERAL LINEAR MODELS PROCEDURE

DEPENDENT VARIABLE: HPM        PGMR-MNGR MAN-MONTHS

SOURCE                DF      SUM OF SQUARES       MEAN SQUARE      F VALUE        PR > F      R-SQUARE          C.V.

MODEL                  1      77302.11838437    77302.11838437       240.46        0.0001      0.919681       37.9057

ERROR                 21       6751.09674644      321.48079745                    STD DEV                    HPM MEAN

UNCORRECTED TOTAL     22      84053.21513081                                  17.92988559                 47.30129046


SOURCE                DF           TYPE I SS      F VALUE      PR > F      DF         TYPE IV SS      F VALUE      PR > F

DEV_PROD               1      77302.11938437       240.46      0.0001       1     77302.11838437       240.46      0.0001


                                   T FOR HO:      PR > |T|          STD ERROR OF
PARAMETER            ESTIMATE     PARAMETER=0                        ESTIMATE

DEV_PROD           1.28860722           15.51       0.0001          0.08310026
```

Table A-6. Log-Linear Model for Programmer and Manager Staff-Months

```
                                        GENERAL LINEAR MODELS PROCEDURE
DEPENDENT VARIABLE: HPMT

SOURCE                 DF     SUM OF SQUARES       MEAN SQUARE      F VALUE         PR > F       R-SQUARE           C.V.

MODEL                   1        24.85438952       24.85438952       256.36         0.0001       0.927631         9.2909

ERROR                  20         1.93901974        0.09695099                     STD DEV                     HPMT MEAN

CORRECTED TOTAL        21        26.79340926                                     0.31136953                   3.35133205

SOURCE                 DF          TYPE I SS      F VALUE      PR > F       DF      TYPE IV SS      F VALUE       PR > F

DEVPRODT                1        24.85438952       256.36      0.0001        1     24.85438952       256.36       0.0001

                                  T FOR HO:       PR > |T|          STD ERROR OF
PARAMETER            ESTIMATE    PARAMETER=0                         ESTIMATE

INTERCEPT          0.59541255           3.23       0.0042          0.18448168
DEVPRODT           0.92090572          16.01       0.0001          0.05751614
```

Table A-7. Non-Linear Model for Total Staff-Months

```
NON-LINEAR LEAST SQUARES SUMMARY STATISTICS     DEPENDENT VARIABLE HPMO .

      SOURCE                DF      SUM OF SQUARES        MEAN SQUARE

      REGRESSION             2     107172.82145158     53586.41072579
      RESIDUAL              20       6124.86385210       306.24319261
      UNCORRECTED TOTAL     22     113297.68530368

      (CORRECTED TOTAL)     21      47220.23066245

PARAMETER          ESTIMATE        ASYMPTOTIC               ASYMPTOTIC 95 %
                                   STD. ERROR             CONFIDENCE INTERVAL
                                                        LOWER            UPPER
A                4.02840216        1.67798979       0.52820317       7.52860115
B                0.76802759        0.09771563       0.56419789       0.97185729

                  ASYMPTOTIC CORRELATION MATRIX OF THE PARAMETERS

                                          A           B

                          A        1.000000   -0.993068
                          B       -0.993068    1.000000

NOTE: ALL ASYMPTOTIC STATISTICS ARE APPROXIMATE. REFERENCE: RALSTON AND JENNRICH, TECHNOMETRICS, FEBRUARY 1978, P 7-14.
```

Table A-8. Linear Model for Total Staff-Months

```
                                         GENERAL LINEAR MODELS PROCEDURE

DEPENDENT VARIABLE: HPMD        TOTAL MAN-MONTHS

SOURCE                  DF      SUM OF SQUARES       MEAN SQUARE      F VALUE        PR > F     R-SQUARE          C.V.

MODEL                    1     105536.98707900   105536.98707900       285.58        0.0001     0.931502       35.0772

ERROR                   21       7760.69822468      369.55705832                    STD DEV                 HPMD MEAN

UNCORRECTED TOTAL       22     113297.68530368                                 19.22386689               54.80438546


SOURCE                  DF          TYPE I SS      F VALUE      PR > F      DF      TYPE IV SS      F VALUE      PR > F

DEV_PROD                 1     105536.98707900      285.58      0.0001       1   105536.98707900     285.58      0.0001


                                     T FOR HO:       PR > |T|        STD ERROR OF
PARAMETER              ESTIMATE     PARAMETER=0                        ESTIMATE

DEV_PROD             1.50566103           16.90       0.0001          0.08909752
```

Table A-9. Log-Linear Model for Total Staff-Months

```
                                      GENERAL LINEAR MODELS PROCEDURE

DEPENDENT VARIABLE: HPMOT

SOURCE              DF     SUM OF SQUARES       MEAN SQUARE     F VALUE        PR > F       R-SQUARE          C.V.

MODEL                1        24.82436572       24.82436572      258.09        0.0001       0.928082        8.8637

ERROR               20         1.92367636        0.09618382                   STD DEV                    HPMOT MEAN

CORRECTED TOTAL     21        26.74804207                                  0.31013516                    3.49892904

SOURCE              DF          TYPE I SS    F VALUE     PR > F        DF       TYPE IV SS      F VALUE      PR > F

DEVPRODT             1        24.82436572     258.09     0.0001         1      24.82436572       258.09      0 0001

                                T FOR HO:    PR > |T|         STD ERROR OF
PARAMETER           ESTIMATE  PARAMETER=O                      ESTIMATE

INTERCEPT         0.74467460         4.05      0.0006          0.18375033
DEVPRODT          0.92034933        16.07      0.0001          0.05728813
```

REFERENCES

1. Software Engineering Laboratory, SEL-81-104, The Software Engineering Laboratory, D. N. Card, F. E. McGarry, G. Page, et al., February 1982
2. V. R. Basili, "Models and Metrics for Software Management and Engineering, "ASME Advances in Computer Technology, January 1980, Vol. 1
3. J. W. Bailey and V. R. Basili, "A Meta-Model for Software Development Resource Expenditures," Proceedings of the Fifth International Conference on Software Engineering. New York: Computer Societies Press, 1981
4. M. L. Ralston and R. I. Jennrich, "Dud, A Derivative-Free Algorithm for Nonlinear Least Squares," Technometrics, February 1978, Vol. 20, No. 1.
5. SAS Institute, Statistical Analysis System User's Guide, J. H. Goodnight, J. P. Sall, J. T. Helwig, et al., 1979

D4-61

N87-24899

EARLY ESTIMATION OF RESOURCE EXPENDITURES AND PROGRAM SIZE

Prepared by

COMPUTER SCIENCES CORPORATION

D. Card

For

GODDARD SPACE FLIGHT CENTER

Under

Contract NAS 5-24300

June 1982

1. INTRODUCTION

A substantial amount of software engineering research effort has been focused on the development of software cost estimation models. A concensus (of sorts) has emerged on that topic. The following relationship is widely accepted:

$$H_s = aL^b \qquad (1)$$

where H_s = staff-hours of effort
L = lines of code
a = a constant
b = a constant

The Software Engineering Laboratory (SEL) has devised a measure of lines of code based on the origin of the delivered code that is substituted in the equation above. This is

$$L_{dev} = N + E + 0.2\ (S+O) \qquad (2)$$

where L_{dev} = "developed" lines of code
N = newly implemented lines of code
E = extensively modified lines of code
S = slightly modified lines of code
O = old (unchanged) lines of code

Equation 1 using "developed" lines of code has given good results as an estimator of development effort. (The analyses in this document are based on a sample of 20 ground-based attitude systems). Table 13 shows a regression analysis that produced a correlation of 0.99 and an estimate of b of 1.1 when the value of a was fixed at 1.0 in Equation 1. Despite these encouraging results, this model has two sigificant limitations. These are the following:

- The substantial amount of development work done in activities other than code implementation may not be adequately considered in the lines of code measure.

- The lines of code, whether "delivered" or "developed", is not known accurately until late in the development cycle when accurate estimates are less useful.

The purpose of this memorandum is to discuss these limitations and to propose some alternative estimation models that can be used earlier in the development process, e.g., during requirements analysis and preliminary design.

2. MODELS OF WORK

The obvious alternative to lines of code as a measure of the work done is pages of documentation. Although only a portion of a software development team is involved in coding, almost everyone produces some documentation. This includes requirements, design, and operations documents. Table 1 compares the components of developed lines of code with pages of documentation as estimators or programmer hours. A regession model based on the two most strongly correlated measures is described in Table 2. This model showed the following relationship:

$$H_p = 0.056\ N + 4.15D \qquad (3)$$

where H_p = programmer hours
N = newly implemented lines of code
D = pages of documentation

A similar comparison is made in Table 3 for these measures as estimators of staff-hours (including programmer, manager, and other hours). A regression model based on the two most strongly correlated measures is described in Table 4. This model showed the following relationship:

$$H_s = 0.051\ N + 7.10D \qquad (4)$$

where H_s = staff-hours
N = newly implemented lines of code
D = pages of documentation

The correlation coefficient (r) associated with each of the relationships expressed in Equations 3 and 4 was 0.97, comparable to that obtained by substituting Equation 2 for L in Equation 1. These results suggest that the best measures of work done are lines of new code and pages of documentation. Reused lines of code do not seem to contribute directly to resource expenditures. However, the requirements analysis and design effort involved in reusing previously developed code may be included in the pages of documentation measure.

Although pages of documentation appears to be an important measure of work, it has the same limitation as lines-of-code measures. Pages of documentation cannot be determined accurately early in the development cycle. The next sections discusses some other measures that can be used to develop models for early estimation of resource expenditures and program size.

3. MODELS FOR EARLY ESTIMATION

Few objective measures are available early in the software development process. The following five measures were considered in this analysis:

- Number of subsystems - requirements analysis
- Number of data sets - preliminary design
- Complexity (PRICE-S) - preliminary design
- Number of new modules - detailed design
- Number of reused modules (extensively modified, slightly modified, and old) - detailed design

The following sections discuss the use of these measures for early estimation of program size and resource expenditures.

3.1 PROGRAM SIZE

The correlations of the measures described here with delivered lines of code are compared in Table 5. Three regression models were developed (Tables 6, 7, and 8). The two most useful of these are the following:

$$L_{del} = 7596\ S \qquad (5)$$

$$L_{del} = 168N + 195R \qquad (6)$$

where L_{del} = delivered lines of code
S = number of subsystems
N = number of new modules
R = number of reused modules

Equation 5 ($r = 0.99$) defines an estimating relationship for program size that can be used during the requirements analysis phase. Equation 6 ($r = 0.98$) defines an estimating relationship of comparable reliability that can be used during the design phase.

3.2 RESOURCE EXPENDITURES

The correlations of the measures described here with staff-hours of effort are compared in Table 9. Three regression models were developed (Tables 10, 11, and 12). The two most useful of these are the following:

$$H_s = 1634\ S \qquad (7)$$

$$H_s = 45\ N + 28\ R \qquad (8)$$

where H_s = staff-hours
S = number of subsystems
N = number of new modules
R = number of reused modules

Equation 7 ($r = 0.93$) defines an estimating relationship for resource expenditures that can be used during the requirements analysis phase. Equation 8 ($r = 0.94$) defines an estimating relationship of higher reliability that can be used during the design phase.

4. CONCLUSION

The preceding analysis has demonstrated two important points. These are the following:

- New measures of productivity which incorporate other development products besides lines of code must be investigated. Pages of documentation is a good candidate.
- Effective estimates of program size and resource expenditures can be made using measures that are available early in the development cycle.

Table 1. Components of Programmer Effort

```
N=      20     REGRESSION MODELS FOR DEPENDENT VARIABLE PGMRHRS

NUMBER IN      R-SQUARE     VARIABLES IN MODEL
  MODEL

    1         0.08943231     OLDLINES
    1         0.49044658     MODLINES
    1         0.80450662     NEWLINES
    1         0.85046601     DOCPAGES
-----------------------------------------
    2         0.49386580     MODLINES OLDLINES
    2         0.80450674     NEWLINES MODLINES
    2         0.81581865     NEWLINES OLDLINES
    2         0.85129683     OLDLINES DOCPAGES
    2         0.85198581     MODLINES DOCPAGES
    2         0.85887286     NEWLINES DOCPAGES
--------------------------------------------------
    3         0.81685888     NEWLINES MODLINES OLDLINES
    3         0.85257247     MODLINES OLDLINES DOCPAGES
    3         0.85888650     NEWLINES OLDLINES DOCPAGES
    3         0.86233681     NEWLINES MODLINES DOCPAGES
-----------------------------------------------------------
    4         0.86261078     NEWLINES MODLINES OLDLINES DOCPAGES
--------------------------------------------------------------------
```

Table 2. Model of Programmer Effort

```
                                      GENERAL LINEAR MODELS PROCEDURE

DEPENDENT VARIABLE: PGMRHRS
SOURCE                  DF        SUM OF SQUARES         MEAN SQUARE      F VALUE          PR > F       R-SQUARE           C.V.

MODEL                    2   106607575032.508650  53303787516.254325       130.92          0.0001       0.935679        36.2358

ERROR                   18     7328452413.491348    407136245.193964                      STD DEV                  PGMRHRS MEAN

UNCORRECTED TOTAL       20   113936027446.000000                                     20177.617431                 55684.20000000

SOURCE                  DF           TYPE I SS     F VALUE      PR > F      DF          TYPE IV SS      F VALUE      PR > F

NEWLINES                 1   103353493284.406580      253.85      0.0001       1     448211536.116894         1.10      0.3080
DOCPAGES                 1     3254081748.102075        7.99      0.0112       1    3254081748.102075         7.99      0.0112

                                    T FOR HO:     PR > |T|      STD ERROR OF
PARAMETER               ESTIMATE    PARAMETER=0                   ESTIMATE

NEWLINES              0.55992183          1.05      0.3080        0.53364909
DOCPAGES             41.51867299          2.83      0.0112       14.68585102
```

Table 3. Components of Total Staff Effort

```
N=      20      REGRESSION MODELS FOR DEPENDENT VARIABLE MANHRS

NUMBER IN       R-SQUARE      VARIABLES IN MODEL
  MODEL

    1          0.09707264      OLDLINES
    1          0.53222266      MODLINES
    1          0.81364699      NEWLINES
    1          0.88750508      DOCPAGES
-------------------------------------
    2          0.53593783      MODLINES OLDLINES
    2          0.81530674      NEWLINES MODLINES
    2          0.82758155      NEWLINES OLDLINES
    2          0.88779376      MODLINES DOCPAGES
    2          0.88803206      OLDLINES DOCPAGES
    2          0.89026281      NEWLINES DOCPAGES
----------------------------------------------
    3          0.82762271      NEWLINES MODLINES OLDLINES
    3          0.88823599      MODLINES OLDLINES DOCPAGES
    3          0.89028560      NEWLINES OLDLINES DOCPAGES
    3          0.89106130      NEWLINES MODLINES DOCPAGES
-------------------------------------------------------
    4          0.89106286      NEWLINES MODLINES OLDLINES DOCPAGES
------------------------------------------------------------------
```

Table 4. Model of Total Staff Effort

```
                                        GENERAL LINEAR MODELS PROCEDURE

DEPENDENT VARIABLE: MANHRS

SOURCE                 DF      SUM OF SQUARES         MEAN SQUARE      F VALUE         PR > F      R-SQUARE           C.V.

MODEL                   2  238607979283.655180  119303989641.827590     167.97         0.0001      0.949144        32.3398

ERROR                  18   12784812713.344818     710267372.963601                   STD DEV                  MANHRS MEAN

UNCORRECTED TOTAL      20  251392791997.000000                                  26650.841881                  82408.75000000

SOURCE                 DF            TYPE I SS      F VALUE     PR > F       DF        TYPE IV SS      F VALUE      PR > F

NEWLINES                1  229088029089.200590       322.54     0.0001        1    370237194.893545       0.52      0.4796
DOCPAGES                1    9519950194.454598        13.40     0.0018        1   9519950194.454598      13.40      0.0018

                                         T FOR HO:      PR > |T|        STD ERROR OF
PARAMETER                ESTIMATE      PARAMETER=0                        ESTIMATE

NEWLINES               0.50889234             0.72        0.4796         0.70485019
DOCPAGES              71.01442466             3.66        0.0018        19.39725019
```

Table 5. Comparison of Early Size Estimators

```
N=      20      REGRESSION MODELS FOR DEPENDENT VARIABLE TOTLINES

NUMBER IN      R-SQUARE      VARIABLES IN MODEL
  MODEL

    1         0.05087892     COMPLXTY
    1         0.60752653     NODATSET
    1         0.68352123     REMODS
    1         0.77270837     NEWMOD
    1         0.96967766     NOSUBSYS
-----------------------------------
    2         0.60988357     NODATSET COMPLXTY
    2         0.71154503     REMODS COMPLXTY
    2         0.77923131     NODATSET NEWMOD
    2         0.79114322     NEWMOD COMPLXTY
    2         0.87902381     NODATSET REMODS
    2         0.93248454     NEWMOD REMODS
    2         0.97280856     NOSUBSYS NODATSET
    2         0.97356977     NOSUBSYS NEWMOD
    2         0.97363300     NOSUBSYS COMPLXTY
    2         0.97737439     NOSUBSYS REMODS
------------------------------------------
    3         0.79403377     NODATSET NEWMOD COMPLXTY
    3         0.88406426     NODATSET REMODS COMPLXTY
    3         0.93555143     NEWMOD REMODS COMPLXTY
    3         0.93928611     NODATSET NEWMOD REMODS
    3         0.97487138     NOSUBSYS NODATSET NEWMOD
    3         0.97550227     NOSUBSYS NEWMOD COMPLXTY
    3         0.97689986     NOSUBSYS NODATSET COMPLXTY
    3         0.97746439     NOSUBSYS NEWMOD REMODS
    3         0.97757499     NOSUBSYS NODATSET REMODS
    3         0.97907109     NOSUBSYS REMODS COMPLXTY
---------------------------------------------------
    4         0.94078488     NODATSET NEWMOD REMODS COMPLXTY
    4         0.97742335     NOSUBSYS NODATSET NEWMOD COMPLXTY
    4         0.97764845     NOSUBSYS NODATSET NEWMOD REMODS
    4         0.97907134     NOSUBSYS NEWMOD REMODS COMPLXTY
    4         0.97955179     NOSUBSYS NODATSET REMODS COMPLXTY
------------------------------------------------------------
    5         0.97956096     NOSUBSYS NODATSET NEWMOD REMODS COMPLXTY
--------------------------------------------------------------------
```

Table 6. Minimal Size Estimating Model

```
                                          GENERAL LINEAR MODELS PROCEDURE

DEPENDENT VARIABLE: TOTLINES

SOURCE                  DF       SUM OF SQUARES        MEAN SQUARE       F VALUE        PR > F       R-SQUARE          C.V.

MODEL                    1   47483674672.6913720  47483674672.6913720      1143.24        0.0001       0.983652       17.5551

ERROR                   19     789152506.3086271    41534342.4372962                     STD DEV                TOTLINES MEAN

UNCORRECTED TOTAL       20   48272827179.0000000                                       6444.7143022              36711.35000000

SOURCE                  DF            TYPE I SS      F VALUE       PR > F         DF           TYPE IV SS       F VALUE       PR > F

NOSUBSYS                 1   47483674672.6913720      1143.24       0.0001          1  47483674672.6913690      1143.24       0.0001

                                     T FOR HO:       PR > |T|           STD ERROR OF
PARAMETER                ESTIMATE  PARAMETER=0                             ESTIMATE

NOSUBSYS            7595.77764277        33.81         0.0001           224.64861839
```

Table 7. Optimal Size Estimating Model

```
                                     GENERAL LINEAR MODELS PROCEDURE

DEPENDENT VARIABLE: TOTLINES

SOURCE               DF      SUM OF SQUARES         MEAN SQUARE      F VALUE       PR > F      R-SQUARE          C.V.

MODEL                 2  46817506873.1303250  23408753436.5651620     289.53       0.0001      0.969852       24.4930

ERROR                18   1455320305.8696746     80851128.1038708                 STD DEV                TOTLINES MEAN

UNCORRECTED TOTAL    20  48272827179.0000000                                    8991.7255354              36711.35000000

SOURCE               DF              TYPE I SS      F VALUE      PR > F      DF          TYPE IV SS     F VALUE     PR > F

NEWMOD                1  43332342186.0297470       535.95      0.0001       1     6521075210.1470580       80.66     0.0001
REMODS                1   3485164687.1005773        43.11      0.0001       1     3485164687.1005768       43.11     0.0001

                                     T FOR HO:       PR > |T|         STD ERROR OF
PARAMETER               ESTIMATE    PARAMETER=0                         ESTIMATE

NEWMOD              168.14286708         8.98       0.0001          18.72241579
REMODS              195.10551281         6.57       0.0001          29.71672397
```

Table 8. Alternative Size Estimating Model

```
                                         GENERAL LINEAR MODELS PROCEDURE

DEPENDENT VARIABLE: TOTLINES

SOURCE                 DF        SUM OF SQUARES           MEAN SQUARE      F VALUE          PR > F       R-SQUARE           C.V.

MODEL                   3    47824276670.3418870   15941425556.7806290      604.18          0.0001       0.990708        13.8920

ERROR                  17      448550508.6581125      26385324.0387125                     STD DEV                  TOTLINES MEAN

UNCORRECTED TOTAL      20    48272827179.0000000                                      5136.6646804                 36711.35000000


SOURCE                 DF             TYPE I SS      F VALUE      PR > F       DF           TYPE IV SS      F VALUE       PR > F

NOSUBSYS                1    47483674672.6913720      1799.62      0.0001        1   5853376833.9617080       221.84       0.0001
REMODS                  1      177236696.2114050         6.72      0.0190        1    130771745.4506511         4.96       0.0398
COMPLXTY                1      163365301.4391105         6.19      0.0235        1    163365301.4391105         6.19       0.0235


                                        T FOR HO:      PR > |T|        STD ERROR OF
PARAMETER              ESTIMATE       PARAMETER=0                        ESTIMATE

NOSUBSYS           7408.97828817            14.89        0.0001         497.43495025
REMODS               52.08554530             2.23        0.0398          23.39599237
COMPLXTY            -46.45837911            -2.49        0.0235          18.67090462
```

Table 9. Comparison of Early Resource Estimators

```
N=      20      REGRESSION MODELS FOR DEPENDENT VARIABLE MANHRS

NUMBER IN       R-SQUARE      VARIABLES IN MODEL
  MODEL

     1        0.01590804      COMPLXTY
     1        0.45501952      REMODS
     1        0.45541937      NODATSET
     1        0.70892650      NEWMOD
     1        0.72646057      NOSUBSYS
-----------------------------------
     2        0.45626141      NODATSET COMPLXTY
     2        0.46118614      REMODS COMPLXTY
     2        0.61892671      NODATSET REMODS
     2        0.71178663      NODATSET NEWMOD
     2        0.72655608      NOSUBSYS REMODS
     2        0.72877159      NOSUBSYS NODATSET
     2        0.74250148      NOSUBSYS COMPLXTY
     2        0.74906820      NOSUBSYS NEWMOD
     2        0.76095094      NEWMOD. COMPLXTY
     2        0.76680965      NEWMOD REMODS
-----------------------------------------
     3        0.61906266      NODATSET REMODS COMPLXTY
     3        0.72903802      NOSUBSYS NODATSET REMODS
     3        0.74310346      NOSUBSYS REMODS COMPLXTY
     3        0.74504992      NOSUBSYS NODATSET COMPLXTY
     3        0.76091707      NOSUBSYS NODATSET NEWMOD
     3        0.76689486.     NOSUBSYS NEWMOD REMODS
     3        0.76956103      NODATSET NEWMOD REMODS
     3        0.77156956      NODATSET NEWMOD COMPLXTY
     3        0.78545331      NOSUBSYS NEWMOD COMPLXTY
     3        0.80024180      NEWMOD REMODS COMPLXTY
--------------------------------------------------
     4        0.74860158      NOSUBSYS NODATSET REMODS COMPLXTY
     4        0.77089041      NOSUBSYS NODATSET NEWMOD REMODS
     4        0.80025824      NOSUBSYS NEWMOD REMODS COMPLXTY
     4        0.80557302      NOSUBSYS NODATSET NEWMOD COMPLXTY
     4        0.80898615      NODATSET NEWMOD REMODS COMPLXTY
-----------------------------------------------------------
     5        0.81071231      NOSUBSYS NODATSET NEWMOD REMODS COMPLXTY
--------------------------------------------------------------------
```

Table 10. Minimal Resource Estimating Model

```
                                          GENERAL LINEAR MODELS PROCEDURE

DEPENDENT VARIABLE: MANHRS

SOURCE                  DF      SUM OF SQUARES        MEAN SQUARE      F VALUE        PR > F      R-SQUARE         C.V.

MODEL                    1  219779472655.727810  219779472655.727810     132.09        0.0001      0.874247      49.4977

ERROR                   19   31613319341.272186    1663858912.698536                  STD DEV                MANHRS MEAN

UNCORRECTED TOTAL       20  251392791997.000000                                   40780.426729              82408.75000000

SOURCE                  DF           TYPE I SS      F VALUE     PR > F        DF        TYPE IV SS     F VALUE      PR > F

NOSUBSYS                 1  219779472655.727810       132.09     0.0001         1  219779472655.727790      132.09      0.0001

                                        T FOR HO:       PR > |T|         STD ERROR OF
PARAMETER               ESTIMATE      PARAMETER=0                         ESTIMATE

NOSUBSYS          16341.56500608            11.49         0.0001         1421.86489246
```

Table 11. Optimal Resource Estimating Model

```
                                        GENERAL LINEAR MODELS PROCEDURE

DEPENDENT VARIABLE: MANHRS

SOURCE                  DF      SUM OF SQUARES        MEAN SQUARE      F VALUE        PR > F      R-SQUARE          C.V.

MODEL                    2   224429018553.875590  112214509276.937780       74.91        0.0001      0.892742       46.9657

ERROR                   18    26963773443.124404    1497987413.506911                   STD DEV                MANHRS MEAN

UNCORRECTED TOTAL       20   251392791997.000000                                    38703.842361              82408.75000000


SOURCE                  DF           TYPE I SS      F VALUE      PR > F         DF         TYPE IV SS      F VALUE      PR > F

NEWMOD                   1   217210830613.300650       145.00      0.0001          1   47581329259.352713        31.76      0.0001
REMODS                   1     7218187940.574946         4.82      0.0415          1    7218187940.574943         4.82      0.0415

                                          T FOR HO:      PR > |T|          STD ERROR OF
PARAMETER                 ESTIMATE      PARAMETER=0                          ESTIMATE

NEWMOD                454.18954107             5.64        0.0001          80.58847287
REMODS                280.78375723             2.20        0.0415         127.91220057
```

Table 12. Alternative Resource Estimating Model

```
                                        GENERAL LINEAR MODELS PROCEDURE

DEPENDENT VARIABLE: MANHRS

SOURCE                   DF      SUM OF SQUARES        MEAN SQUARE      F VALUE          PR > F       R-SQUARE          C.V.

MODEL                     3  224740240779.482830  74913413593.160930      47.78          0.0001       0.893980        48.0476

ERROR                    17   26652551217.517166   1567797130.442186                    STD DEV                 MANHRS MEAN

UNCORRECTED TOTAL        20  251392791997.000000                                    39595.418049               82408.75000000

SOURCE                   DF          TYPE I SS     F VALUE     PR > F        DF         TYPE IV SS     F VALUE     PR > F

NEWMOD                    1  217210830613.300650    138.55     0.0001         1   35681483401.478013     22.76     0.0002
REMODS                    1    7218187940.574946      4.60     0.0466         1    7506110162.185972      4.79     0.0429
COMPLXTY                  1     311222225.607251      0.20     0.6615         1     311222225.607251      0.20     0.6615

                                       T FOR HO:      PR > |T|          STD ERROR OF
PARAMETER               ESTIMATE     PARAMETER=0                          ESTIMATE

NEWMOD               479.86369347           4.77      0.0002          100.58688857
REMODS               289.54047766           2.19      0.0429          132.32647759
COMPLXTY             -62.79840646          -0.45      0.6615          140.94778457
```

Table 13. Log Model of Resource-Size Relationship

```
                                          GENERAL LINEAR MODELS PROCEDURE
DEPENDENT VARIABLE: MANHRS

SOURCE                 DF      SUM OF SQUARES      MEAN SQUARE      F VALUE        PR > F        R-SQUARE          C.V.

MODEL                   1      2348.55070494     2348.55070494     19095.41        0.0001        0.999006        3.2512

ERROR                  19         2.33681641        0.12299034                    STD DEV                   MANHRS MEAN

UNCORRECTED TOTAL      20      2350.88752135                                   0.35069978                   10.78669231

SOURCE                 DF         TYPE I SS      F VALUE      PR > F       DF        TYPE IV SS      F VALUE      PR > F

DEVLINES                1      2348.55070494     19095.41      0.0001        1     2348.55070494     19095.41      0.0001

                                  T FOR HO:       PR >.|T|          STD ERROR OF
PARAMETER            ESTIMATE    PARAMETER=0                          ESTIMATE

DEVLINES           1.10558402         138.19        0.0001           0.00800069
```

Table 14. SUMMARY STATISTICS FOR MEASURES

MEASURE	MEAN	MEDIAN
DOCPAGES	995	762
NEWLINES	25,928	13,550
TOTLINES	36,711	16,265
PGMRHRS	5,568.4	3,051.1
MANHRS	8,240.9	4,541.8
NEWMOD	133	88
REMODS	76	32

Figure 1. Relationship of Modules to Size

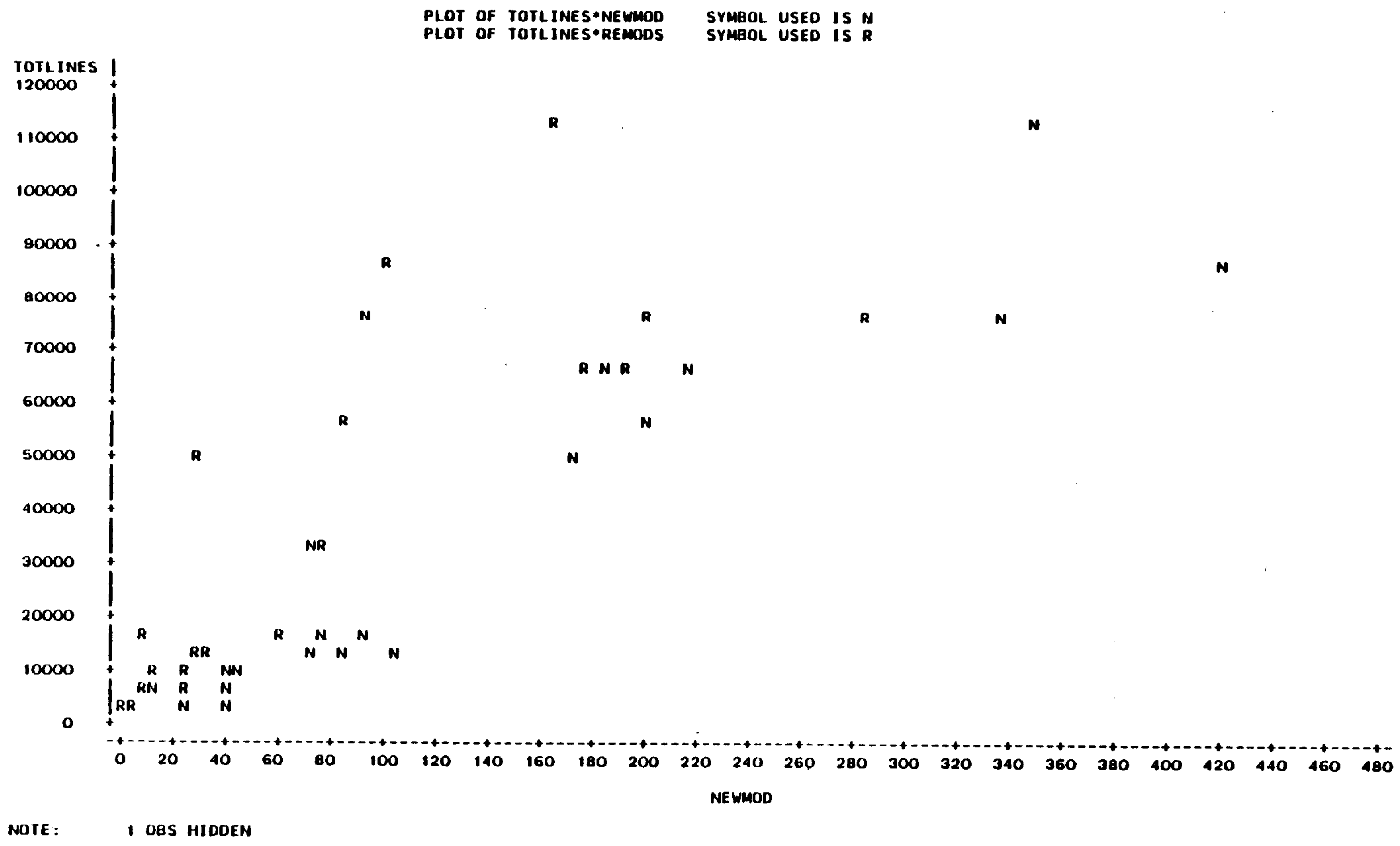
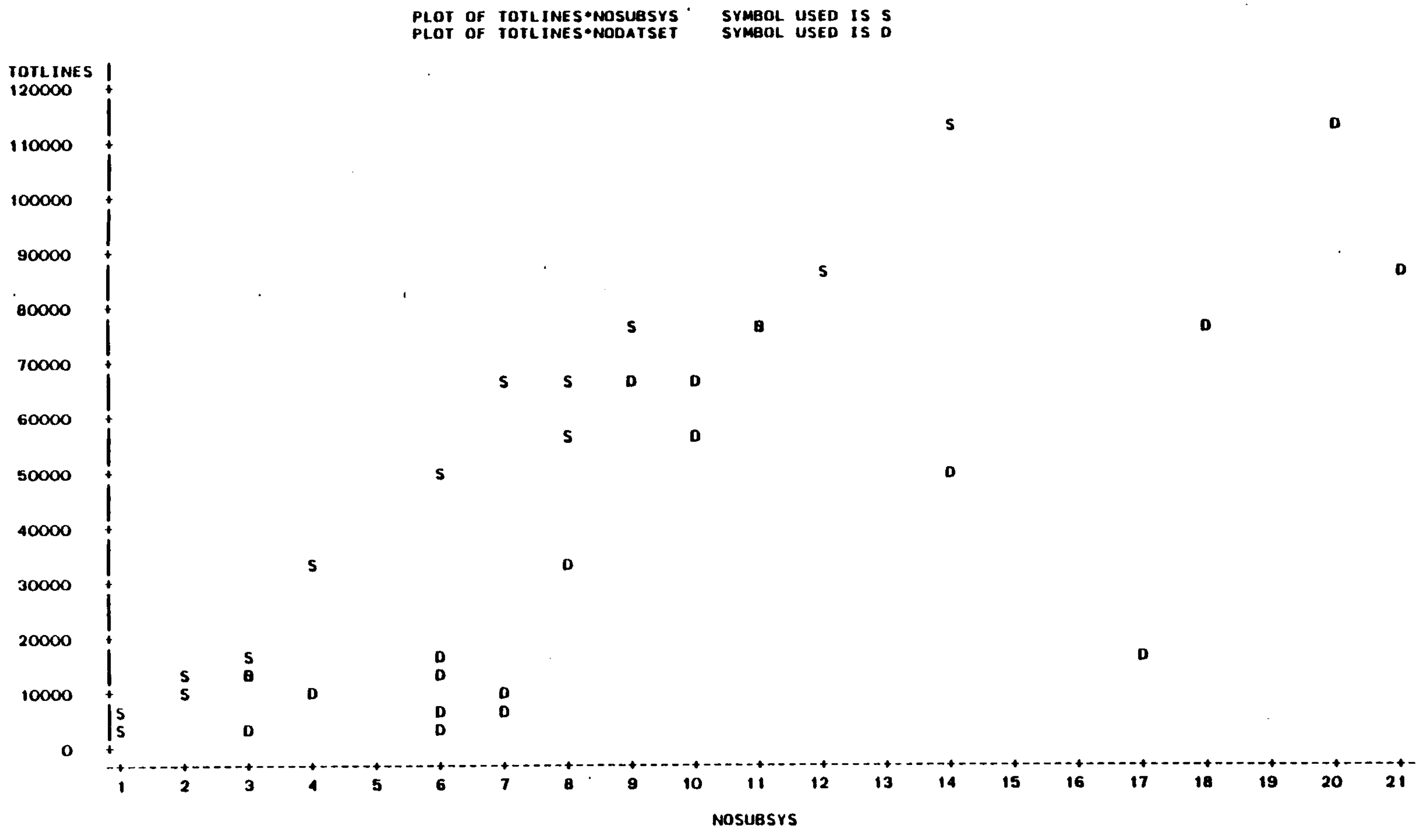


Figure 2. Relationship of System to Size



NOTE: 6 OBS HIDDEN

Figure 3. Relationship of Modules to Total Staff Effort

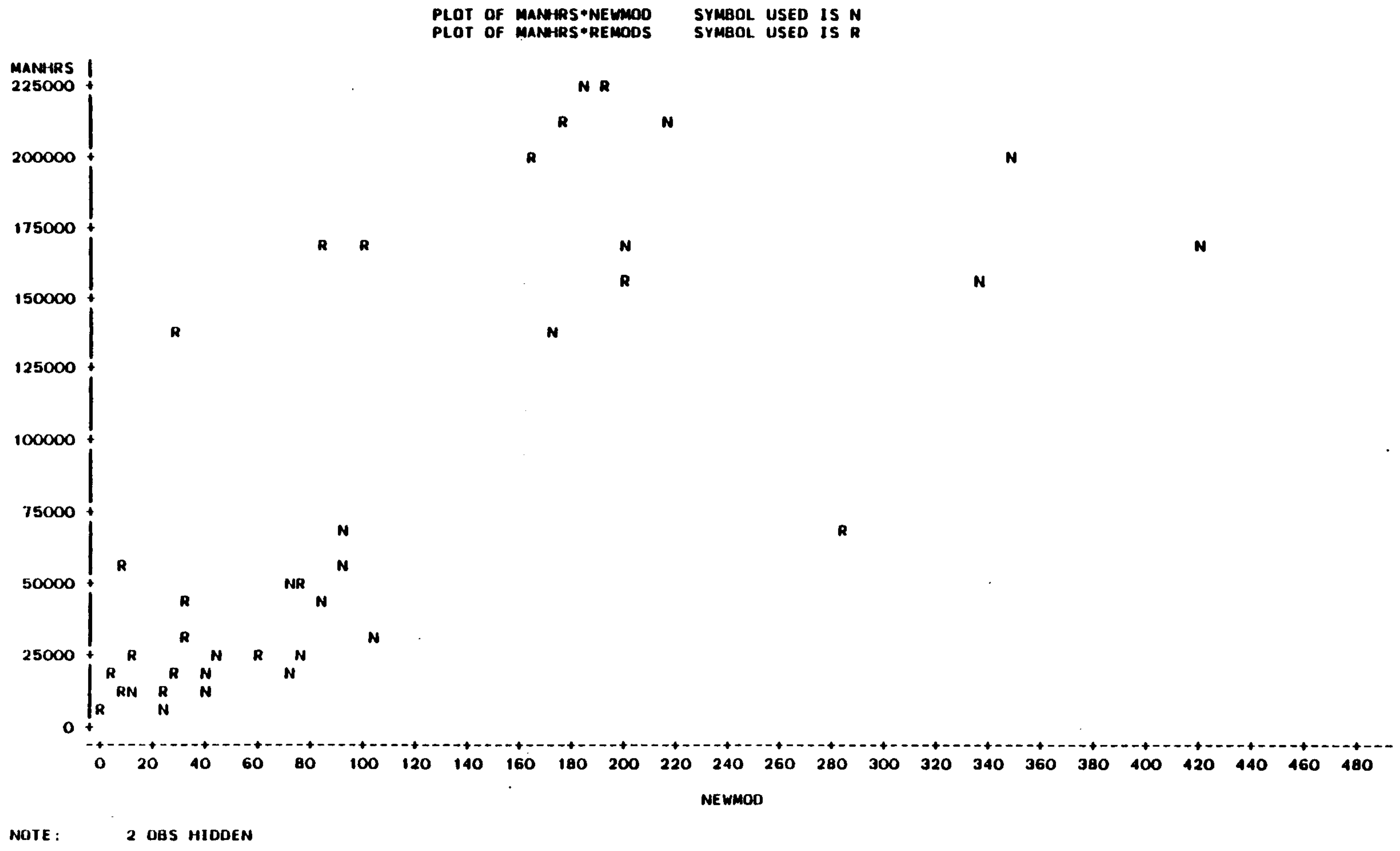
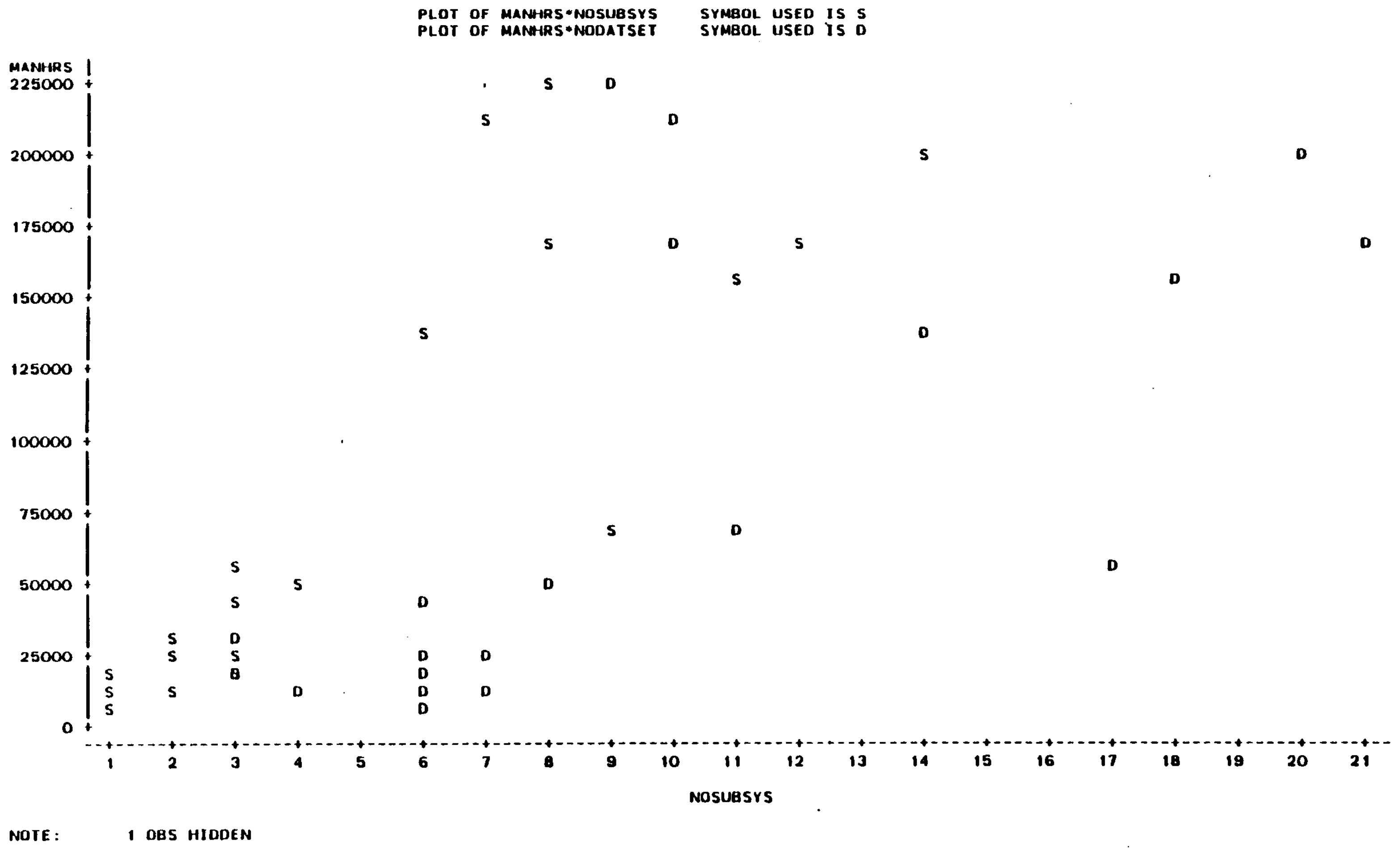


Figure 4. Relationship of System to Total Staff Effort



SECTION 4 – SOFTWARE MEASURES

omit

SECTION 4 - SOFTWARE MEASURES

The technical papers included in this section were originally published as indicated below.

- Basili, V. R., R. W. Selby, and T. Phillips, "Metric Analysis and Data Validation Across FORTRAN Projects," University of Maryland, Technical Report TR 1228, November 1982 (reprinted by permission of the authors)

  A version of this paper also appears in IEEE Transactions on Software Engineering, November 1983, vol. 9, no. 7.

- Doerflinger, C. W., and V. R. Basili, "Monitoring Software Development Through Dynamic Variables," University of Maryland, Technical Memorandum, August 1983 (reprinted by permission of the authors).

  A version of this paper also appears in Proceedings of the Seventh International Computer Software and Applications Conference. New York: Computer Societies Press, November 1983.

- Basili, V. R., and B. T. Perricone, "Software Errors and Complexity: An Empirical Investigation, "University of Maryland, Technical Report TR-1195, August 1982 (reprinted by permission of the authors)

  A version of this paper will appear in Communications of the ACM, January 1984, vol. 27, no. 1.

D5-61

N87-24900

Technical Report TR-1228

November 1982
NSG-5123
AFOSR-F49620-80-C-001

METRIC ANALYSIS AND DATA VALIDATION ACROSS FORTRAN PROJECTS *

Victor R. Basili, Richard W. Selby, Jr.
and Tsai-Yun Phillips

Department of Computer Science
University of Maryland
College Park, MD 20742

*Research supported in part by the National Aeronautics and Space Administration Grant NSG-5123 and the Air Force Office of Scientific Research Contract AFOSR-F49620-80-C-001 to the University of Maryland. Computer support provided in part by the facilities of NASA/Goddard Space Flight Center and the Computer Science Center at the University of Maryland.

ABSTRACT

The desire to predict the effort in developing or explain the quality of software has led to the proposal of several metrics in the literature. As a step toward validating these metrics, the Software Engineering Laboratory has analyzed the Software Science metrics, cyclomatic complexity and various standard program measures for their relation to 1) effort (including design through acceptance testing), 2) development errors (both discrete and weighted according to the amount of time to locate and fix) and 3) one another. The data investigated are collected from a production FORTRAN environment and examined across several projects at once, within individual projects and by individual programmers across projects, with three effort reporting accuracy checks demonstrating the need to validate a database. When the data come from individual programmers or certain validated projects, the metrics′ correlations with actual effort seem to be strongest. For modules developed entirely by individual programmers, the validity ratios induce a statistically significant ordering of several of the metrics′ correlations. When comparing the strongest correlations, neither Software Science′s E metric, cyclomatic complexity nor source lines of code appears to relate convincingly better with effort than the others.

I. Introduction

Several metrics based on characteristics of the software product have appeared in the literature. These metrics attempt to predict the effort in developing or explain the quality of that software [11], [17], [19], [23]. Studies have applied them to data from various organizations to determine their validity and appropriateness [1], [13], [15]. However, the question of how well the various metrics really measure or predict effort or quality is still an issue in need of confirmation. Since development environments and types of software vary, individual studies within organizations are confounded by variations in the predictive powers of the metrics. Studies across different environments will be needed before this question can be answered with any degree of confidence.

Among the most popular metrics have been the Software Science metrics of Halstead [19] and the cyclomatic complexity metric of McCabe [23]. The Software Science E metric attempts to quantify the complexity of understanding an algorithm. Cyclomatic complexity has been applied to establish quality thresholds for programs. Whether these metrics relate to the concepts of effort and quality depends on how these factors are defined and measured. The definition of effort employed in this paper is the amount of time required to produce the software product (the number of man-hours programmers and managers spent from the beginning of functional design to the end of acceptance testing). One aspect of software quality is the number of errors reported during the product's development, and this is the measure associated with quality for this study.

Regarding a metric evaluation, there are several issues that need to be addressed. How well do the various metrics predict or explain these measures of effort and quality? Does the correspondence increase with greater accuracy of effort and error reporting? How do these metrics compare in predictive power to simpler and more standard metrics, such as lines of source code or the number of executable statements? These questions deal with the external validation of the metrics. More fundamental questions exist dealing with the internal validation or consistency of the metrics. How well do the estimators defined actually relate to the Software Science metrics? How do the Software Science metrics, the cyclomatic complexity metric and the more traditional metrics relate to one another? In this paper, both sets of issues are addressed. The analysis examines whether the given family of metrics is internally consistent and attempts to determine how well these metrics really measure the quantities that they theoretically describe.

One goal of the Software Engineering Laboratory [6], [7], [8], [10], a joint venture between the University of Maryland, NASA/Goddard Space Flight Center and Computer Sciences Corporation, has been to provide an experimental database for examining these relationships and providing insights into the answering of such questions.

The software comprising the database is ground support software for satellites. The systems analyzed consist of 51,000 to 112,000 lines of FORTRAN source code and took between 6900 and 22,300 man-hours to develop over a period of 9 to 21 months. There are from 200 to 600 modules (e.g., subroutines) in each system and the staff size ranges from 8 to 23 people, including the support personnel. While anywhere from 10 to 61 percent of the source code is modified from previous projects, this analysis focuses on just the newly developed modules.

The next section discusses the data collection process and some of the potential problems involved. The third section defines the metrics and interprets the counting procedure used in their calculation. In the fourth section, the Software Science metrics are correlated with their estimators and related to more primitive program measures. Finally, the fifth section determines how well this collection of volume and complexity metrics corresponds to actual effort and developmental errors.

II. The Data

The Software Engineering Laboratory collects data that deal with many aspects of the development process and product. Among these data are the effort to design, code and test the various modules of the systems as well as the errors committed during their development. The collected data are analyzed to provide insights into software development and to study the effect of various factors on the process and product. Unlike the typical controlled experiments where the projects tend to be smaller and the data collection process dominates the development process, the major concern here is the software development process, and the data collectors must affect minimal interference to the developers.

This creates potential problems with the validity of the data. For example, suppose we are interested in the effort expended on a particular module and one programmer forgets to turn in his weekly effort report. This can cause erroneous data for all modules the programmer may have worked on that week. Another problem is how does a programmer report time on the integration testing of three modules? Does he charge the time to the parent module of all three, even though that module may be just a small driver? That is clearly easier to do than to proportion the effort between all three modules he has worked on. Another issue is how to count errors. An error that is limited to one module is easy to assign. What about an error that required the analysis of ten modules to determine that it affects changes in three modules? Does the programmer associate one error with all ten modules, an error with just the three modules or one third of an error with each of the three?~ The larger the system

~ Efforts [18], [21] have attempted to make this assignment scheme more precise by the explanation: a "fault" is a specific manifestation in the source code of a programmer "error"; due to a misconception or document discrepancy, a programmer commits an "error" that can result in several "faults" in the program. With this interpretation, what are referred to as errors in this study should probably be called faults. In the interest of consistency with previous work and clarity, however, the term error will be used throughout the paper.

the more complicated the association. All this assumes that all the errors are reported. It is common for programmers not to report clerical errors because the time to fill out the error report form might take longer than the time to fix the error. These subtleties exist in most observation processes and must be addressed in a fashion that is consistent and appropriate for the environment.

The data discussed in this paper are extracted from several sources. Effort data were obtained from a Component Status Report that is filled out weekly by each programmer on the project. They report the time they spend on each module in the system partitioned into the phases of design, code and test, as well as any other time they spend on work related to the project, e.g., documentation, meetings, etc. A module is defined as any named object in the system; that is, a module is either a main procedure, block data, subroutine or function. The Resource Summary Form, filled out weekly by the project management, represents accounting data and records all time charged to the project for the various personnel, but does not break effort down on a module basis. Both of these effort reports are utilized in Section V of this paper to validate the effort reporting on the modules. The errors are collected from the Change Report Forms that are completed by a programmer each time a change is made to the system. While the collection of effort and error data is a subjective process and done manually, the remainder of the software measures are objective and their calculation is automated.

A static code analyzing program called SAP [25] automatically computes several of the metrics examined in this analysis. On a module basis, the SAP program determines the number of source and executable statements, the cyclomatic complexity, the primitive Software Science metrics and various other volume and complexity related measures. Computer Sciences Corporation developed SAP specifically for the Software Engineering Laboratory and the program has been recently updated [14] to incorporate a more consistent and thorough counting scheme of the Software Science parameters. In an earlier study, Basili and Phillips [3] employed the preliminary version of SAP in a related analysis. The next section explains the revised counting procedure and defines the various metrics.

III. Metric Definition

In the application of each of the metrics, there exist various ways to count each of the entities. This section interprets the counting procedure used by the updated version of SAP and defines each of the metrics examined in the analysis. These definitions are given relative to the FORTRAN language, since that is the language used in all the projects studied here. The counting scheme depends on the syntactic analysis performed by SAP and is, therefore, not necessarily chosen to coincide exactly with other definitions of the various counts.

<u>Primitive Software Science metrics</u> Software Science defines the vocabulary metric n as the sum of the number of unique operators n1 and the number of unique operands n2. The operators fall into three classes.

i) Basic operators include

```
+   -   *   /   **   =   ()   &   //   .NE.   .EQ.   .LE.   .LT.
.GE.   .GT.   .AND.   .OR.   .XOR.   .NOT.   .EQV.   .NEQV.
```

ii) Keyword operators include

```
IF() THEN                   /* logical if */
IF() THEN ELSE              /* logical if-then-else */
IF()   , ,                  /* arithmetic if */
IF() THEN ENDIF             /* block if */
IF() THEN ELSE ENDIF        /* block if-then-else */
IF() THEN
   ELSEIF() THEN
    ...   ENDIF             /* case if */
DO                          /* do loop */
DOWHILE                     /* while loop */
GOTO <target>               /* unconditional goto: distinct
                               targets imply different operators */
GOTO (T1...Tn) <expr>       /* computed goto: different number of
                               targets imply different operators */
GOTO <ident>, (T1...Tn)     /* assigned goto: distinct identifiers
                               imply different operators */
<subr>( , ,*<target>)       /* alternate return */
END=                        /* read/write option */
ERR=                        /* read/write option */
ASSIGNTO                    /* target assignment */
EOS                         /* implicit statement delimiter */
```

iii) Special operators consist of the names of subroutines, functions and entry points.

Operands consist of the all variable names and constants. Note that the major differences of this counting scheme from that used by Basili and Phillips [3] are in the way goto and if statements are counted.

The metric n* represents the potential vocabulary, and Software Science defines it as the sum of the minimum number of operators n1* and the minimum number of operands n2*. The potential operator count n1* is equal to two; that is, n1* equals one grouping operator plus one subroutine/function designator. In this paper, the potential operand count n2* is equal to the sum of the number of variables referenced from common blocks, the number of formal parameters in the subroutine and the number of additional arguments in entry points.

Source lines This is the total number of source lines that appear in the module, including comments and any data statements while excluding blank lines.

Source lines - comments This is the difference between the number of source lines and the number of comment lines.

Executable statements This is the number of FORTRAN executable statements that appear in the program.

Cyclomatic complexity Cyclomatic complexity is defined as being the number of partitions of the space in a module's control-flow graph. For programs with unique entry and exit nodes, this metric is equivalent to one plus the number of decisions and in this work, is equal to the one plus sum of the following constructs: logical if's, if-then-else's, block-if's, block if-then-else's, do loops, while loops, AND's, OR's, XOR's, EQV's, NEQV's, twice the number of arithmetic if's, n - 1 decision counts for a computed goto with n statement labels and n decision counts for a case if with n predicates.

A variation on this definition excludes the counts of AND´s, OR´s, XOR´s, EQV´s and NEQV´s (later referred to as Cyclo_cmplx_2).

<u>Calls</u> This is the number of subroutine and function invocations in the module.

<u>Calls and jumps</u> This is the total number of calls and decisions as they are defined above.

<u>Revisions</u> This is the number of versions of the module that are generated in the program library.

<u>Changes</u> This is the total number of changes to the system that affected this module. Changes are classified into the following types (a single change can be of more than one type):

a. error correction
b. planned enhancement
c. implement requirements change
d. improve clarity
e. improve user service
f. debug statement insertion/deletion
g. optimization
h. adapt to environment change
i. other

<u>Weighted changes</u> This is a measure of the total amount of effort spent making changes to the module. A programmer reports the amount of effort to actually implement a given change by

indicating either

a. less than one hour,
b. one hour to a day,
c. one day to three days or
d. over three days.

The respective means of these durations, 0.5, 4.5, 16 and 32 hours, are divided equally among all modules affected by the change. The sum of these effort portions over all changes involving a given module defines the weighted changes for the module.

Errors This is the total number of errors reported by programmers; i.e., the number of system changes that listed this module as involved in an error correction. (See the footnote at the bottom of page 4 regarding the usage of the term "error".)

Weighted errors This is a measure of the total amount of effort spent isolating and fixing errors in a module. For error corrections, a programmer also reports the amount of effort spent isolating the error by indicating either

a. less than one hour,
b. one hour to one day,
c. more than one day or
d. never found.

The representative amounts of time for these durations, 0.5, 4.5, 16 and 32 hours, are combined with the effort to implement the correction (as calculated earlier) and divided equally among the modules changed. The sum of these effort portions over all error corrections involving a given module defines the weighted errors for the module.

IV. Internal Validation of the Software Science Metrics

The purpose of this section is to briefly define the Software Science metrics, to see how these metrics relate to standard program measures and to determine if the metrics are internally consistent. That is, Software Science hypothesizes that certain estimators of the basic parameters, such as program length N and program level L, can be approximated by formulas written totally in terms of the number of unique operators and operands. Initially, an attempt is made to find correlations between various definitions of these quantities based on the interpretations of operators and operands given in the previous section. Then, the family of metrics that Software Science proposes is correlated with traditional measures of software.

Program length Program length N is defined as the sum of the total number of operators N1 and the total number of operands N2; i.e., $N = N1 + N2$. Software Science hypothesizes that this can be approximated by an estimator N^ that is a function of the vocabulary, defined as

$$\hat{N} = n1 \log2(n1) + n2 \log2(n2).$$

The scatter plot appearing in Figure 1 and Pearson correlation coefficient of .899 ($p < .001$; 1794 modules)~ show the relationship between N and N^ (polynomial regression rejects including a second degree term at $p = .05$). Several sources [12], [16], [26], [27] have observed that the length estimator tends to be

~ The symbol p will be used to stand for significance level.

high for small programs and low for large programs. The correlations and significance levels for the pairwise Wilcoxon statistic [20], broken down by executable statements and length, are displayed in Table 1. In our environment, either measure of size demonstrates that N^ significantly overestimates N in the first and second quartiles and underestimates it (most significantly) in the fourth quartile. Feuer and Fowlkes [15] assert that the accuracy of the relation between the natural logarithms of estimated and observed length changes less with program size. The scatter plot appearing in Figure 2 and correlation coefficient for ln N vs. ln N^ of .927 ($p < .001$; 1794 modules) show moderate improvement.

<< Figure 1 >>

<u>Table 1</u>. <u>Observed vs. estimated length broken down by program size</u>.

a. N vs. N^ broken down by executable statments.

XQT STMTS	MODS	R~	ESTIMATION	WILCOXON SIGNIF
0 - 19	446	.601	over	<<.0001
20 - 40	442	.511	over	<<.0001
41 - 78	457	.478	under	.0367
79 <=	449	.751	under	<<.0001

b. N vs. N^ broken down by N.

Length N	MODS	R~	ESTIMATION	WILCOXON SIGNIF
0 - 114	449	.750	over	<<.0001
115 - 243	445	.447	over	<<.0001
244 - 512	453	.348	under	.0010
513 <=	447	.731	under	<<.0001

~ ($p < .001$)

<< Figure 2 >>

C-2

<u>Program volume</u> A program volume metric V defined as N log2 n represents the size of an implementation, which can be thought of as the number of bits necessary to express it. The potential volume V* of an algorithm reflects the minimum representation of that algorithm in a language where the required operation is already defined or implemented. The parameter V* is a function of the number of input and output arguments of the algorithm and is meant to be a measure of its specification. The metric V* is defined as

$$V^* = (2 + n2^*) \log2 (2 + n2^*).$$

The correlation coefficient for V vs. V* of .670 ($p < .001$; 1794 modules) shows a reasonable relationship between a program's necessary volume and its specification.

<u>Program level</u> The program level L for an algorithm is defined as the ratio of its potential volume to the size of its implementation, expressed as

$$L = V^*/V.$$

Thus, the highest level for an algorithm is its program specification and there L has value unity. The larger the size of the required implementation V, the lower the program level of the implementation. Since L requires the calculation of V*, which is not always readily obtainable, Software Science hypothesizes that L can be approximated by

$$L\hat{} = \frac{2\ n2}{n1\ N2} .$$

The correlation for L vs. $L^{\wedge}$ of .531 ($p < .001$; 1794 modules) is disappointingly below that of .90 given in [19]. Hoping for an increase in the correlations, the modules are partitioned by the number of executable statements in Table 2. Although the upper quartiles show measured improvement over the correlation of the whole sample, a more interesting relationship surfaces. The level estimator significantly underestimates the program level in the second, third and fourth quartiles, with the hypothesis being rejected in the first quartile. The increase in magnitude of the n2* parameter does not appear to be totally captured by the definition of $L^{\wedge}$.

Table 2. Relationship of observed vs. estimated program level broken down by program size.

XQT STMTS	MODS	R~	ESTIMATION	WILCOXON SIGNIF
0 - 19	446	.484	--	--
20 - 40	442	.672	under	<<.0001
41 - 78	457	.597	under	<<.0001
79 <=	449	.615	under	<<.0001
all	1794	.531	under	<<.0001

~ ($p < .001$)

Program difficulty The program difficulty D is defined as the difficulty of coding an algorithm. The metric D and the program level L have an inverse relationship; D is expressed

$$D = 1/L .$$

An alternate interpretation of difficulty defines it as the inverse of $L^{\wedge}$, given by

$$D2 = \frac{1}{\hat{L}} = \frac{n1N2}{2\ n2} \quad .$$

Christensen, Fitsos and Smith [12] demonstrate that the unique operator count n1 tends to remain relatively constant with respect to length for 490 PL/S programs. They propose that the average operand usage N2/n2 is the main contributor to the program difficulty D2. The scatter plot appearing in Figure 3 and Pearson correlation coefficient of .729 ($p < .001$; 1794 modules) display the relationship between N2/n2 and D2 for our FORTRAN modules. The application of polynomial regression brings in a second degree term ($p < .001$) and results in a correlation of .738.

<< Figure 3 >>

However, after observing in Figure 4 that n1 varies with program size, it seems as if the n1's inflation might possibly better explain D2. The scatter plot appearing in Figure 5 and the correlation of .865 ($p < .001$; 1794 modules) show the relationship of D2 vs. n1. Step-wise polynomial regression brings in a second degree term initially, followed by a linear term ($p <$ .001), and results in a correlation of .879. In our environment, the unique operator count n1 explains a greater proportion of the variance of the difficulty D2 than the average operand usage N2/n2.

<< Figure 4 >>

<< Figure 5 >>

Program effort The Software Science effort metric E attempts to quantify the effort required to comprehend the implementation of an algorithm. It is defined as the ratio of the volume of an implementation to its level, expressed as

$$E = \frac{V}{L} = \frac{(V)^{2}}{V^*} .$$

The E metric increases for programs implemented with large volumes or written at low program levels; that is, it varies with the square of the volume. An approximation to E can be obtained without the knowledge of the potential volume by substituting L^ for L in the above equation. The metric

$$E\hat{} = \frac{V}{L\hat{}} = \frac{n1\ N2\ V}{2\ n2} = \frac{n1\ N2\ N\ \log 2\ n}{2\ \ n2}$$

defines the product of one half the number of unique operators, the average operand usage and the volume. In an attempt to remove the effect of possible program impurities [9], [19], N^ is substituted for N in the above equation, yielding

$$E\hat{}\hat{} = \frac{N\hat{}\ \log 2\ n}{L\hat{}} = \frac{n1\ N2\ (n1\log 2n1 + n2\log 2n2)\ \log 2\ n}{2\ \ n2} .$$

The correlation coefficients for E vs. E^, E vs. E^^, ln E vs. ln E^ and ln E vs. ln E^^ are given in Table 3a. A fit of a least squares regression line to the log-log plot of E vs. E^ produces the equation

$$\ln E = .830 * \ln \hat{E} + 1.357 .$$

Equivalently,

$$E = \exp(1.357) * (\hat{E})**0.830 .$$

Due to this non-linear relationship and the improved correlation of ln E vs. ln E^, the modules are partitioned by executable statements in Table 3b. The application of polynomial regression confirms this non-linearity by bringing in a second degree term ($p < .001$), resulting in a correlation of .698. In Table 3b, notice that the correlations seem substantially better for modules below median size. The significant overestimation in the upper three quartiles attributes to the relationship of L and L^ described earlier.

<u>Table 3</u>. <u>Observed vs. estimated Software Science E metric</u>.

<u>a</u>. <u>Pearson Correlation</u> (<u>p</u> < .<u>001</u>; <u>1794 modules</u>).

	R
E vs. E^	.663
ln E vs. ln E^	.931
E vs. E^^	.603
ln E vs. ln E^^	.890

<u>b</u>. <u>E vs. E^ broken down by executable statements</u>.

XQT STMTS	MODS	R~	ESTIMATION	WILCOXON SIGNIF
0 - 19	446	.708	under	.0050
20 - 40	442	.709	over	<<.0001
41 - 78	457	.411	over	<<.0001
79 <=	449	.550	over	<<.0001

~ ($p < .001$)

<u>Program bugs</u> Software Science defines the bugs metric B as the total number of "delivered" bugs in a given implementation. Not to be confused with user acceptance testing, the metric B is

the number of inherent errors in a system component at the completion of a distinct phase in its development. Bugs B is expressed by

$$B = L \frac{E}{Eo} = \frac{V}{Eo}$$

where Eo is theoretically equivalent to the mean number of elementary discriminations between potential errors in programming. Through a calculation that employs the definitions of E, L and lambda (lambda = LV* is referred to as the language level), this equation becomes

$$B = \frac{(lambda)^{**}1/3\ (E)^{**}2/3}{Eo} .$$

The derivation determines an Eo value of 3000, assumes (lambda)**1/3 ~= 1 and obtains

$$B\hat{} = \frac{(E)^{**}2/3}{3000} .$$

The correlation for B vs. B^ is .789 ($p < .001$; 1794 modules).

In summary, the relationship of some of the Software Science metrics with their estimators seems to be program size dependent. Several observations lead to the result that the metric N^ significantly overestimates N for modules below the median size and underestimates for those above the median size. The level estimator L^ seems to have a moderate correlation with L, and its significant underestimation of L in the upper three quartiles reflects its failure to capture the magnitude of n2* in the larger modules. With respect to the E metric, the effort estimator E^ correlates better over the whole sample than E^^, and their strongest correlations are for modules below median size. The estimator E^ shows a non-linear relationship to the effort metric E. The correlation of ln E vs. ln E^ significantly improves over that of E vs. E^, with the E^ metric's overestimation of E for larger modules attributing to the role of L^ in its definition. With the above family of metrics, Software Science attempts to quantify size and complexity related concepts that have traditionally been described by a more fundamental set of measures.

Table 4 displays the correlations of the Software Science metrics with the classical program measures of source lines of code, cyclomatic complexity, etc. There are several observations worth noting. Length N and volume V have remarkably similar correlations and correspond quite well with most of the program measures. Several of the metrics correlate well with the number of executable statements, especially the program "size" metrics of N1, N2, N and V (also B). The level estimator L^ and its inverse D2 seem to be much more related to the standard size and complexity measures than their counterparts L and D1. The language level lambda does not seem to show a significant relationship to the standard size and complexity measures, as expected. The E^^ metric relates best with the number of executable statements and the modified cyclomatic complexity, while correlating with all the measures better than the E metric and slightly better than E^. None of the Software Science measures correlate especially well with the number of revisions or the sum

Table 4. Comparison of Software Science metrics against more traditional software measures.

Key:	
?	not significant at .05 level
*	significant at .05 level
a	significant at .01 level
otherwise	significant at .001 level

	Source_Lines	Execut_Stmts	Source-Cmmts	Cyclo_cmplx	Cyclo_cmplx_2	Revisions	Calls_&_Jumps	Calls
n1	.776	.854	.778	.796	.818	.361	.802	.542
n2	.852	.867	.853	.767	.774	.430	.809	.614
N1	.824	.964	.868	.881	.889	.328	.869	.552
N2	.826	.949	.871	.858	.870	.355	.870	.597
n2*	.792	.691	.754	.635	.629	.501	.683	.541
N	.829	.961	.873	.874	.884	.343	.874	.577
N^	.864	.897	.864	.800	.811	.420	.836	.621
V -	.837	.962	.875	.873	.883	.343	.876	.584
V*	.776	.677	.734	.618	.611	.485	.664	.525
L	-.098	-.179	-.112	-.170	-.173	?	-.158	-.083
L^	-.383	-.411	-.394	-.389	-.396	-.216	-.386	-.250
D1=1/L	.067a	.244	.113	.178	.196	-.093	.134	?
D2=1/L^	.696	.872	.745	.816	.839	.269	.791	.478
N2/n2	.365	.544	.437	.508	.517	.106	.470	.241
Lambda	.136	?	.108	?	?	.134	?	.051*
E	.439	.629	.500	.535	.556	.106	.506	.282
E^	.663	.831	.711	.771	.797	.224	.748	.452
E^^	.738	.871	.760	.799	.829	.268	.788	.501
B -	.837	.962	.875	.873	.883	.343	.876	.584
B^	.546	.749	.610	.650	.670	.149	.620	.355

- B and V will have identical correlations since they are linear functions of one another.

of procedure and function calls. The primary measures of unique operators n1 and unique operands n2 correspond reasonably well overall with n2 being stronger with source lines and n1 stronger with the cyclomatic complexities. In the next section, an analysis attempts to determine the relationship that these parameters really have with the quantities that they theoretically describe.

V. External Validation of the Software Science and Related Metrics

The purpose of this section is to determine how well the Software Science metrics and various complexity measures relate to actual effort and errors encountered during the development of software in a commercial environment. These objective product metrics are compared against more primitive volume metrics, such as lines of source code. The reservoir of development data includes the monitoring of several projects and the analysis examines several projects at once, individual projects and individual programmers across projects. To remove the dependency of the distribution of the correlation coefficient on the actual measures of effort and errors, the nonparametric Spearman rank order correlation coefficients are examined in this section [22]. (The ability of a few data points to artificially inflate or deflate the Pearson product-moment correlation coefficient is well recognized.) The analysis first examines how well these measures correspond to the total effort spent in the development of software.

A. Metrics' Relation to Actual Effort

Initially, a correlation across seven projects of the Software Science E metric vs. actual effort, on a module by module basis using only those that are newly developed, produces the results in Table 5. The table also displays the correlations of some of the more standard volume metrics with actual effort. These disappointingly low correlations create a fear that there

Table 5. Spearman rank order correlations Rs with effort for all modules (731) from all projects.

Key:	
?	not significant at .05 level
*	significant at .05 level
a	significant at .01 level
otherwise	significant at .001 level

E	.345
E^	.445
E^^	.488
Cyclo_cmplx	.463
Cyclo_cmplx_2	.467
Calls	.414
Calls_&_Jumps	.494
D1=1/L	.126
D2=1/L^	.417
Source_Lines	.522
Execut_Stmts	.456
Source-Cmmts	.460
V	.448
N	.434
eta1	.485
eta2	.461
B	.448
B^	.345
Revisions	.531
Changes	.469
Weighted_Chg	.468
Errors	.220
Weighted_Err	.226

may be some modules with poor effort reporting skewing the analysis. Since there is partial redundancy built into the effort data collection process, there exists hope of validating the effort data.

<u>Validation of effort data</u> The partial redundancy in the development monitoring process is that both managers and programmers submit effort data. Individual programmers record time spent on each module, partitioned by design, code, test and support phases, on a weekly basis with a Component Status Report (CSR). Managers record the amount of time every programmer spends working each week on the project they are supervising with a Resource Summary Form (RSF). Since the latter form possesses the enforcement associated with the distribution of financial resources, it is considered more accurate [24]. However, the Resource Summary Form does not break effort down by module, and thus a combination of the two forms has to be used.

Three different possible effort reporting validity checks are proposed. All employ the idea of selecting programmers that tend to be good effort reporters, and then using just the modules that only they worked on in the metric analysis. The three proposed effort reporting validity checks are:

a. $$Vm = \frac{\text{number of weekly CSR's submitted by programmer}}{\text{number of weeks programmer appears on RSF's}}$$

b. $$Vt = \frac{\text{sum of all man-hours reported by programmer on all CSR's}}{\text{sum of all man-hours reported for programmer on all RSF's}}$$

c. $$Vi = 1 - \frac{\text{number of weeks programmer's CSR effort} > \text{RSF effort}}{\text{total number of weeks programmer active in project}}$$

The first validity proposal attempts to capture the frequency of the programmer's effort reporting. It checks for missing data by ranking the programmers according to the ratio Vm of the number of Component Status Reports submitted over the number of weeks that the programmer appears on Resource Summary Forms. The second validity proposal attempts to capture the total percentage of effort reported by the programmer. This proposal ranks the programmers according to the ratio Vt formed by the sum of all the man-hours reported on Component Status Reports over the sum of all hours delegated to him on Resource Summary Forms.

Note that for a given week, the amount of time reported on a Component Status Report should be always less than or equal to the amount of time reported on the corresponding Resource Summary Form. This is not because the programmer fails to "cover" himself, but a consequence of the management's encouragement for programmers to realisticly allocate their time rather than to guess in an ad hoc manner. This observation defines a third validity proposal to attempt to capture the frequency of a programmer's reporting of inflated effort. This data check ranks the programmers according to the quantity Vi equal to one minus the ratio of the number of weeks that CSR effort reported exceeded RSF effort over the total number of weeks that the programmer is active in the project.

Metrics' relation to validated effort data Of the given proposals, the systems development head of the institution where the software is being developed suggests that the first proposal, the missing data check, would be a good initial attempt to select modules with accurate effort reporting [24]. The missing data ratios Vm are defined for programmers on a project by project basis. Table 6 displays the effort correlations of the newly developed modules worked on by only programmers with Vm >= 90% from all projects, those with Vm >= 80% and for all newly developed modules. Most of the correlations of the modules included in the Vm >= 90% level seem to show improvement over those at the Vm >= 80% level. Although this is the desired effect and several of the Vm >= 90% correlations increase over the original values, a majority of the correlations with modules at the Vm >= 80% level are actually lower than their original coefficients. Since the effect of the ratio's screening of the data is inconsistent and the overall magnitudes of the correlations are low, the analysis now examines modules from different projects separately.

Table 6. Spearman rank order correlations Rs with effort for modules across seven projects with various validity levels.

Key:	
?	not significant at .05 level
*	significant at .05 level
a	significant at .01 level
otherwise	significant at .001 level

	Validity ratio Vm (#mods)		
	all(731)	80%(398)	90%(215)
E	.345	.307	.357
E^	.445	.422	.467
E^^	.488	.480	.513
Cyclo_cmplx	.463	.457	.479
Cyclo_cmplx_2	.467	.454	.506
Calls	.414	.360	.402
Calls_&_Jumps	.494	.475	.479
D1=1/L	.126	.088*	?
D2=1/L^	.417	.371	.421
Source_Lines	.522	.519	.501
Execut_Stmts	.456	.429	.475
Source-Cmmts	.460	.420	.439
V	.448	.434	.475
N	.434	.416	.460
eta1	.485	.462	.493
eta2	.461	.467	.503
B	.448	.434	.475
B^	.345	.307	.357
Revisions	.531	.580	.565
Changes	.469	.495	.385
Weighted_Chg	.468	.521	.462
Errors	.220	.381	.205
Weighted_Err	.226	.382	.247

The Spearman correlations of the various metrics with effort for three of the individual projects appear in Table 7.

Table 7. Spearman rank order correlations Rs with effort for various validity rankings of modules from individual projects S1, S3 and S7.

Key:	
?	not significant at .05 level
*	significant at .05 level
a	significant at .01 level
otherwise	significant at .001 level
z	unavailable data

	Project						
	S1			S3⁻		S7⁻⁻	
Validity ratio Vm	all	80%	90%	80%	90%	all	80%
#modules	79	29	20	132	81	127	49
E	.613	.647	.726	.469	.419	.285	.409a
E^	.665	.713	.746	.602	.585	.389	.569
E^^	.700	.747	.798	.638	.640	.430	.567
Cyclo_cmplx	.757	.774	.792	.583	.608	.463	.523
Cyclo_cmplx_2	.764	.785	.787	.609	.664	.491	.523
Calls	.681	.698	.818	.442	.492	.404	.485
Calls_&_Jumps	.776	.813	.822	.594	.619	.488	.569
D1=1/L	.262a	?	?	.156*	?	?	?
D2=1/L^	.625	.681	.745	.507	.442	.377	.499
Source_Lines	.686	.672	.729	.743	.734	.486	.499
Execut_Stmts	.688	.709	.781	.609	.594	.408	.515
Source-Cmmts	.670	.710	.778	.671	.654	.416	.471
V	.657	.692	.774	.627	.637	.377	.497
N	.653	.680	.755	.613	.619	.360	.484
eta1	.683	.740	.848	.553	.533	.439	.431
eta2	.667	.701	.747	.643	.698	.365	.445
B	.657	.692	.774	.627	.637	.377	.497
B^	.613	.643	.726	.469	.419	.285	.409a
Revisions	.677	.717	.804	.655	.632	.449	.510
Changes	.687	.645	.760	.672	.639	.238a	.380a
Weighted_Chg	.685	.629	.749	.673	.649	.238a	.256*
Errors	z	z	z	.644	.611	.253a	.438
Weighted_Err	z	z	z	.615	.605	.245a	.276*

⁻ All modules in project S3 were developed by programmers with Vm >= 80%.

⁻⁻ There exist fewer than a significant number of modules developed by programmers with Vm >= 90%.

Although the correlation coefficients vary considerably between and among the projects, the overall improvement in projects S1 and S3 is apparent. Almost every metric's correlation with development effort increases with the more reliable data in projects S1 and S7. When comparing the strongest correlations from the seven individual projects, neither Software Science's E metrics, cyclomatic complexity nor source lines of code relates convincingly better with effort than the others. Note that the estimators of the Software Science E metric, E^ and E^^, appear to show a stronger relationship to actual effort than E.

The validity screening process substantially improves the correlations for some projects, but not all. This observation points toward the existence of project dependent factors and interactions. In an attempt to minimize these intraproject effects, the analysis focuses on individual programmers across projects. Note that Basili and Hutchens [2] also suggest that programmer differences have a large effect on the results when many individuals contribute to a project.

The use of modules developed solely by individual programmers significantly reduces the number of available data points because of the team nature of commercial work. Fortunately, however, there are five programmers who totally developed at least fifteen modules each. The correlations for all modules developed by them and their values of the three proposed validity ratios are given in Table 8. The order of increasing correlation coefficients for a particular metric can be related to the order of

Table 8. Spearman rank order correlations Rs with effort for modules totally developed by five individual programmers.

Key:	
?	not significant at .05 level
*	significant at .05 level
a	significant at .01 level
otherwise	significant at .001 level

	Programmer (#mods)				
	P1(31)	P2(17)	P3(21)	P4(24)	P5(15)
E	.593	?	?	.561a	?
E^	.718	.526*	.375*	.555a	.507*
E^^	.789	.570a	?	.539a	.511*
Cyclo_cmplx	.592	.469*	.521a	.565a	?
Cyclo_cmplx_2	.684	.583a	.481*	.546a	?
Calls	.622	.787	?	.669	?
Calls_&_Jumps	.701	.604a	.451*	.579a	?
D1=1/L	.314*	?	?	?	?
D2=1/L^	.713	.460*	?	.497a	.467*
Source_Lines	.863	.682	.605a	.624	?
Execut_Stmts	.747	.540*	.436*	.631	.534*
Source-Cmmts	.826	.576a	.530a	.612	.509*
V	.718	.540*	.453*	.579a	.451*
N	.676	.526*	.461*	.556a	.471*
eta1	.811	.575a	?	.536a	?
eta2	.765	.701	.527a	.597	?
B	.718	.540*	.453*	.579a	.451*
B^	.593	?	?	.561a	?
Revisions	.675	.523*	.777	.468*	?
Changes	.412*	.468*	.600a	?	?
Weighted_Chg	.428a	.527*	.502a	?	?
Errors	.386*	?	.668	?	.596a
Weighted_Err	.342*	?	.624	?	.545*

VALIDITY RATIOS (%)

Vm	92.5	96.0	87.7	83.9	74.1
Vt	97.9	91.8	98.8	82.1	74.1
Vi	78.6	69.5	77.6	80.0	87.5
Ave. Vm,Vt	95.2	93.9	93.25	83.0	74.1
Ave. Vm,Vi	85.5	82.75	82.65	81.95	80.8

increasing values for a given validity ratio using the Spearman rank order correlation. The significance levels of these rank order correlations for several of the metrics appear in Table 9. The statistically significant correspondence between the programmers' validity ratios Vm and the correlation coefficients justifies the use of the ratio Vm in the earlier analysis; possible improvement is suggested if Vm were combined with either of the other two ratios.

Table 9. Significance levels for the Spearman rank order correlation between the programmer's validity ratios and the correlati coefficients for several of the metrics.

Metric	Vm	Vt	Vi	Ave(Vm,Vt)	Ave(Vm,Vi)	Ave(Vt
			Ratio			
E^^				.09	.09	
Cyclo_cmplx						.05
Cyclo_cmplx_2	.05			.02	.02	
Calls_&_Jumps	.05			.02	.02	
Source_Lines	.05			.02	.02	
Source-Cmmts				.09	.09	
V (B)				.09	.09	
eta2	.05			.02	.02	
Revisions		.001	.09^{-}	.09	.09	

$^{-}$ Negative correlation.

In summary, the strongest sets of correlations occur between the metrics and actual effort for certain validated projects and for modules totally developed by individual programmers. While relationships across all projects using both all modules and only validated modules produce only fair coefficients, the validation process shows patterns of improvement. Applying the validity ratio screening to individual projects seems to filter out some of the project specific interactions while not affecting others, with the correlations improving accordingly. Two averages of the validity ratios (Vm with Vt and Vm with Vi) impose a ranking on the individual programmers that statistically agrees with an ordering of the improvement of several of the correlations. In all sectors of the analysis, the inclusion of L^ in the Software Science E metric in its estimators E^ and E^^ seems to improve the metric correlations with actual effort. The analysis now attempts to see how well these metrics relate to the number of errors encountered during the development of software.

B. Metric's Relation to Errors

This section attempts to determine the correspondence of the Software Science and related metrics both to the number of development errors and to the weighted sum of effort required to isolate and fix the errors. A correlation across all projects of the Software Science bugs metric B and some of the standard volume and complexity metrics with errors and weighted errors, using only newly developed modules, produces the results in Table 10. Most of the correlations are very weak, with the exception of system changes. These disappointingly low correlations attribute to the discrete nature of error reporting and that 340 of the 652 modules (52%) have zero reported errors. Even though these correlations show little or no correspondence, the following observations indicate potential improvement.

Table 10. Spearman rank order correlations Rs with errors and weighted-errors for all modules (652) from six projects.

Key:	
?	not significant at .05 level
*	significant at .05 level
a	significant at .01 level
otherwise	significant at .001 level

	Errors	Weighted_err
E	.083*	.101a
E^	.151	.171
E^^	.163	.186
Cyclo_cmplx	.196	.205
Cyclo_cmplx_2	.189	.200
Calls	.220	.236
Calls_&_Jumps	.235	.248
D1=1/L	?	?
D2=1/L^	.124	.140
Source_Lines	.255	.265
Execut_Stmts	.177	.198
Source-Cmmts	.288	.298
V	.168	.186
N	.162	.180
eta1	.102a	.132
eta2	.181	.199
B	.168	.186
B^	.083*	.101a
Revisions	.375	.375
Changes	.677	.636
Weighted_Chg	.627	.677
Design_Eff	.219	.185
Code_Eff	.285	.316
Test_Eff	.149	.164
Tot_Effort	.324	.332

" Project S1 has no data to distinguish errors from changes.

Weiss [4], [5] conducted an extensive error analysis that involved three of the projects and employed enforcement of error reporting through programmer interviews and hand-checks. For two of the more recent projects, independent validation and verification was performed. In addition, the on-site systems development head asserts that due to the maturity of the collection environment, the accuracy of the error reporting is more reliable for the more recent projects [24]. These developmental differences provide the motivation for an examination of the relationships on an individual project basis.

Table 11 displays the attributes of the projects and the correlations of all the metrics vs. errors and weighted errors for three of the individual projects. The correlations in S7, a project involved in the Weiss study, are fair but better than those of project S5 (not shown) that was developed at about the same time. Project S4 and S6 (also not shown) have very poor overall correlations and unreasonably low relationships of revisions with errors, which point to the effect of being early projects in the collection effort. The trend that the attributes produce is not very apparent, although chronology and error reporting enforcement do seem to have some effect. In another attempt to improve the correlations, the analysis applies the

Table 11. Spearman rank order correlations Rs with errors and weighted-errors for modules from three individual projects.

Key: ?	not significant at .05 level
*	significant at .05 level
a	significant at .01 level
otherwise	significant at .001 level
Err	errors
W_err	weighted-errors

	Project (#mods)					
	S3(132)		S4(35)		S7(127)	
	Err	W_err	Err	W_err	Err	W_err
E	.401	.378	?	?	.397	.391
E^	.536	.482	?	?	.507	.503
E^^	.579	.522	?	?	.492	.505
Cyclo_cmplx	.542	.481	?	?	.393	.368
Cyclo_cmplx_2	.553	.489	?	?	.405	.400
Calls	.445	.432	.300*	.316*	.423	.419
Calls_&_Jumps	.566	.518	?	?	.432	.412
D1=1/L	?	?	?	?	.168*	.178*
D2=1/L^	.491	.426	?	?	.563	.559
Source_Lines	.648	.622	.339*	?	.490	.487
Execut_Stmts	.538	.505	?	?	.478	.465
Source-Cmmts	.599	.568	?	?	.501	.483
V	.541	.495	?	?	.461	.456
N	.526	.480	?	?	.457	.449
eta1	.550	.500	?	?	.488	.522
eta2	.541	.500	?	?	.348	.367
B	.541	.495	?	?	.461	.456
B^	.401	.378	?	?	.396	.390
Revisions	.784	.694	.686	.630	.567	.500
Changes	.939	.864	.770	.761	.727	.670
Weighted_Chg	.840	.885	.661	.757	.624	.714
Design_Eff	?	?	?	?	?	?
Code_Eff	.620	.632	.413a	.398a	.274	.264
Test_Eff	.473	.481	.312*	?	?	?
Tot_Effort	.644	.615	.455a	.447a	.253a	.245a

PROJECT ATTRIBUTES

	S3	S4	S7
Weiss study		X	X
IV & V	X		
Chronology	recent	early	middle

previous section's hypothesis of focusing on individual programmers. Table 12 gives the correlations of the metrics with errors and weighted errors for modules that two of the individual programmers totally developed. Even though it is encouraging to see

Table 12. Spearman rank order correlations Rs with errors and weighted-errors for modules totally developed by two individual programmers.

Key:	
?	not significant at .05 level
*	significant at .05 level
a	significant at .01 level
otherwise	significant at .001 level
Err	errors
W_err	weighted-errors

	Programmer (#mods)			
	P2(17)		P3(21)	
	Err	W_err	Err	W_err
E	.514*	.447*	.368*	?
E^	.527*	.493*	.600a	.563a
E^^	.515*	.473*	.666	.649
Cyclo_cmplx	.575a	.558a	.463*	.428*
Cyclo_cmplx_2	.661a	.616a	.484*	.449*
Calls	?	.498*	.506a	.469*
Calls_&_Jumps	.545*	.560a	.598a	.557a
D1=1/L	?	?	?	?
D2=1/L^	.558a	.526*	.459*	.429*
Source_Lines	?	?	.662	.646
Execut_Stmts	.624a	.577a	.579a	.533a
Source-Cmmts	?	.436*	.635	.594a
V	.491*	.472*	.679	.655
N	.494*	.479*	.641	.610a
eta1	.497*	.448*	.611a	.589a
eta2	?	?	.715	.717
B	.491*	.472*	.679	.655
B^	.514*	.447*	.368*	?
Revisions	?	?	.830	.811
Changes	.716	.662a	.855	.828
Weighted_Chg	?	.510*	.863	.861
Design_Eff	?	?	.460*	.392*
Code_Eff	?	.450*	.699	.667
Test_Eff	?	?	.668	.644
Tot_Effort	?	?	.668	.624

the correspondences of the metrics B, E^^ and eta2 with errors as among the best for programmer P3, the same metrics do not relate as well for other programmers.

In summary, partitioning an error analysis by individual project or programmer shows improved correlations with the various metrics. Strong relationships seem to depend on the individual programmer, while few high correlations show up on a project wide basis. The correlations for the projects reflect the positive effects of reporting enforcement and collection process maturity. Overall, the correlations with total errors are slightly higher than those with weighted errors, while the number of revisions appears to relate the best.

VI. Conclusions

In the Software Engineering Laboratory, the Software Science metrics, cyclomatic complexity and various traditional program measures have been analyzed for their relation to effort, development errors and one another. The major results of this investigation are the following: 1) None of the metrics examined seem to manifest a satisfactory explanation of effort spent developing software or the errors incurred during that process; 2) neither Software Science's E metric, cyclomatic complexity nor source lines of code relates convincingly better with effort than the others; 3) the strongest effort correlations are derived when modules obtained from individual programmers or certain validated projects are considered; 4) the majority of the effort correlations increase with the more reliable data; 5) the number of revisions appears to correlate with development errors better than either Software Science's B metric, E metric, cyclomatic complexity or source lines of code; and 6) although some of the Software Science metrics have size dependent properties with their estimators, the metric family seems to possess reasonable internal consistency. These and the other results of this study contribute to the validation of software metrics proposed in the literature. The validation process must continue before metrics can be effectively used in the characterization and evaluation of software and in the prediction of its attributes.

Acknowledgment

The authors are grateful to F. McGarry and B. Curtis for their valuable comments on this analysis. We would also like to thank B. Decker, W. Taylor and E. Edwards for their assistance with the SAP program and the S.E.L. database.

Bibliography

[1] V. R. Basili, Tutorial on Models and Metrics for Software Management and Engineering, IEEE Comput. Society, IEEE Catalog No. EHO-167-7, 1980.

[2] V. R. Basili and D. H. Hutchens, "Analyzing a Syntactic Family of Complexity Metrics," Dept. Comput. Sci., Univ. of Maryland, College Park, MD 20742, Tech. Rep. TR-1053, Dec. 1981 (to appear in T.S.E.).

[3] V. R. Basili and T. Phillips, "Evaluating and Comparing the Software Metrics in the Software Engineering Laboratory," ACM Sigmetrics (1981 ACM Workshop/Symp. Measurement and Evaluation of Software Quality), Vol. 10, pp. 95-106, Mar. 1981.

[4] V. R. Basili and D. M. Weiss, "A Methodology for Collecting Valid Software Engineering Data*," Dept. Comput. Sci., Univ. of Maryland, College Park, MD 20742, Tech. Rep. TR-1235, Dec. 1982.

[5] V. R. Basili and D. M. Weiss, "Evaluating Software Development by Analysis of Changes: The Data from the Software Engineering Laboratory*," Dept. Comput. Sci., Univ. of Maryland, College Park, MD 20742, Tech. Rep. TR-1236, Dec. 1982.

[6] V. R. Basili and M. V. Zelkowitz, "Analyzing Medium Scale Software Developments," Proc. 3rd Int. Conf. Software Eng., Atlanta, GA, May 1978, pp. 116-123.

[7] V. R. Basili and M. V. Zelkowitz, "Measuring Software Development Characteristics in the Local Environment," Computers and Structures, Vol. 10, pp. 39-43, 1979.

[8] V. R. Basili, M. V. Zelkowitz, F. E. McGarry, R. W. Reiter, Jr., W. F. Truszkowski and D. L. Weiss, "The Software Engineering Laboratory," Software Eng. Lab., NASA/Goddard Space Flight Center, Greenbelt, MD 20771, Rep. SEL-77-001, May 1977.

[9] Bulut, Necdet and M. H. Halstead, "Impurities Found in Algorithm Implementations," ACM SIGPLAN Notices, Vol. 9, Mar. 1974.

[10] D. N. Card, F. E. McGarry, J. Page, S. Eslinger and V. R. Basili, "The Software Engineering Laboratory," Software Eng. Lab., NASA/Goddard Space Flight Center, Greenbelt, MD 20771, Rep. SEL-81-104, Feb. 1982.

[11] E. T. Chen, "Program Complexity and Programmer Productivity," IEEE Trans. Software Eng., Vol. SE-4, pp. 187-194, May 1978.

[12] K. Christensen, G. P. Fitsos and C. P. Smith, "A Perspective on Software Science," IBM Syst. J., Vol. 20, pp. 372-387, 1981.

[13] B. Curtis, S. B. Sheppard and P. M. Milliman, "Third Time Charm: Stronger Replication of the Ability of Software Complexity Metrics to Predict Programmer Performance," Proc. 4th Int. Conf. Software Eng., Sept 1979, pp. 356-360.

[14] W. J. Decker and W. A. Taylor, "FORTRAN Static Source Code Analyzer Program (SAP) User's Guide (Revision 1)," Software Eng. Lab., NASA/Goddard Space Flight Center, Greenbelt, MD 20771, Rep. SEL-78-102, May 1982.

[15] A. R. Feuer and E. B. Fowlkes, "Some Results from an Empirical Study of Computer Software," Proc. 4th Int. Conf.

Software Eng., Sept. 1979, pp. 351-355.

[16] G. P. Fitsos, "Vocabulary Effects in Software Science," IBM Santa Teresa Lab., San Jose, CA 95150, Tech. Rep. TR 03.082, Jan. 1980.

[17] J. E. Gaffney and G. L. Heller, "Macro Variable Software Models for Application to Improved Software Development Management," Proc. of Workshop on Quantitative Software Models for Reliability, Complexity and Cost, IEEE Comput. Society, 1980.

[18] S. A. Gloss-Soler, The DACS Glossary: A Bibliography of Software Engineering Terms, Data & Analysis Center for Software, Griffiss Air Force Base, NY 13441, Rep. GLOS-1, Oct. 1979.

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

[20] R. V. Hogg and E. A. Tanis, Probability and Statistical Inference, MacMillian, New York, 1977, pp. 265-271.

[21] IEEE Standard Glossary of Software Engineering Terminology, IEEE, 342 E. 47th St., New York, Rep. IEEE-STD-729-1983, 1983.

[22] M. Kendall and A. Stuart, The Advanced Theory of Statistics, Vol. 2, 4th Ed., MacMillian, New York, 1979, pp. 503-508.

[23] T. J. McCabe, "A Complexity Measure," IEEE Trans. Software Eng., Vol. SE-2, pp. 308-320, Dec. 1976.

[24] F. E. McGarry, Systems Development Head, Code 582.1, NASA/Goddard Space Flight Center, Greenbelt, MD 20771, personal consultation, Jan.-July 1982.

[25] E. M. O'Neill, S. R. Waligora and C. E. Goorevich, "FORTRAN Static Source Code Analyzer (SAP) User's Guide," Software Eng. Lab., NASA/Goddard Space Flight Center, Greenbelt, MD 20771, Rep. SEL-78-002, Feb. 1978.

[26] V. Y. Shen and H. E. Dunsmore, "A Software Science Analysis of COBOL Programs," Dept. Comput. Sci., Purdue Univ., West Lafayette, IN 47907, Tech. Rep. CSD-TR-348, August 1980.

[27] C. P. Smith, "A Software Science Analysis of IBM Programming Products," IBM Santa Teresa Lab., San Jose, CA 95150, Tech. Rep. TR 03.081, Jan. 1980.

D6-61

Monitoring Software Development through Dynamic Variables

N87-24901

ABSTRACT

This paper describes research conducted by the Software Engineering Laboratory (SEL) on the use of dynamic variables as a tool to monitor software development. The intent of the project is to identify project independent measures which may be used in a management tool for monitoring software development. This study examines several FORTRAN projects with similar profiles. The staff was experienced in developing these types of projects. The projects developed serve similar functions. Because these projects are similar we believe some underlying relationships exist that are invariant between the projects. These relationships, once well defined, may be used to compare the development of different projects to determine whether they are evolving the same way previous projects in this environment evolved.

M 915766

Authors:

Carl W. Doerflinger
University of Maryland
Dept. of Computer Science
College Park, MD 20742
(301) 454-4251

Victor R. Basili
University of Maryland
Dept. of Computer Science
College Park, MD 20742
(301) 454-2002

KEYWORDS

management tool, metric, measurement, predictive model

Monitoring Software Development through Dynamic Variables

by

Carl W. Doerflinger
and
Victor R. Basili

I. Overview

The Software Engineering Laboratory (SEL) is a joint effort between the National Aeronautics and Space Administration (NASA), the Computer Sciences Corporation (CSC), and the University of Maryland established to study the software development process. To this end, data has been collected for the last six years. The data was from attitude determination and control software developed by CSC, in FORTRAN, for NASA. Additional information on the SEL, the data collection effort, and some of the studies that have been made may be found in papers from the Software Engineering Laboratory Series published by the SEL [Card82], [Church82], [SEL82].

The interest in the software development process is motivated by a desire to predict costs and quality of projects being planned and developed. For several years, studies have examined the relationships between variables such as effort, size, lines of code, and documentation [Walston77], [Basili81]. These studies, for the most part, used data collected at the end of past projects to predict the behavior of similar projects in the future. In 1981 the SEL concluded that many of these factors were too dependent on the environment to be useful for the models that had been developed [Bailey81]. Any model which attempts to trace these relationships should therefore be calibrated to the environment being examined. The meta-model proposed by the SEL is designed for such flexibility [Bailey81].

Another way to isolate out the environment dependent factors is by comparing two internal factors of a project, thus ignoring all outside influences. One approach that is used to monitor software development examines the time gap between the initial report of software problems and the complete resolution of the problem [Manley82]. Comparing two variables is useful because it also accentuates problem areas as they develop, providing relative information rather than absolute information. Relative information is useful to the project manager because it accentuates trends as the project develops. If project environments are similar, then similar values should be expected. Because the project environments in the SEL are similar, it was felt that this approach could be further extended to provide managers with information about how a set of variables over the course of a project differed from the same set of variables on other projects (baselines). The managers could be alerted to potential problems and use other variable data and project knowledge to determine whether the project was in trouble.

This methodology is flexible enough to respond to changing needs. Every time a project is completed the measures collected during its development may be added in to calculate a new baseline. In this way, the baselines may adapt to any changes in the environment, as they occur.

Baselines might also be developed to reflect different attributes. For instance, several projects which had good productivity might be grouped to form a productivity baseline. Once baselines are established, projects in progress may be compared against them. All measures falling outside the predetermined tolerance range are interpreted by the manager.

II. Methodology

The implementation of this methodology is dependent on two factors. The first factor is the availability of measures that are project independent and can also be collected throughout a project's development. Variables like programmer hours and number of computer runs are project dependent. By comparing these variables against each other a set of relative measures may be generated which is project independent. For instance, the number of software changes may vary from project to project. The project dependent features shared by each variable will cancel out when the ratio of software changes per computer run is taken. The resulting relative measure is project independent.

The second factor is the need for fixed time intervals common to all projects. To normalize for time, project milestones were used. The time into a project might be twenty percent into coding instead of ten weeks into the project, for instance.

When computing the baselines one other factor was considered. At any given interval during development a variable may measure either the total number of events that have occurred from the beginning of development (cumulative) or the number of of events that have occurred since the last measured interval (discrete). Since these approaches may convey different information it was felt that they both should be used.

For simplicity, the baseline for each relative measure was defined as the average and standard deviation computed for the measure at predetermined intervals. A project's progress may now be charted by the software manager. At each interval in a projects development the relative measures are compared with their respective baseline. Any measures outside a standard deviation are flagged. These measures are then interpreted by the project manager to determine how the project is progressing. A flagged measure may indicate a project is developing exceptionally well or it may indicate a problem has been encountered.

The interpretation of a set of flagged measures is a three step process. First, the manager must determine the possible interpretations for each flagged relative measure using lists of possible interpretations developed and verified based on past projects.

Second, the union of the lists of possible interpretations of each flagged measure must be taken. The list formed by this union contains all the possible interpretations ordered using the number of times each interpretation is repeated in the different lists. The larger the number of overlaps a possible interpretation has, the greater the probability it is the correct interpretation.

Third, the manager must analyze the combined list and determine if a problem exists. Interpretations with an equal number of overlaps all have an equal probability of being the correct interpretation. If none of the possible interpretations for a given relative measure overlap then the relative measure should be considered separately.

When analyzing the interpretations, three pieces of information must be considered; the measurements, the point in development, and the managers knowledge of the project. A relative measure may indicate different things depending on the stage of development. For instance, a large amount of computer time per computer run early in the project may indicate not enough unit testing is being done. Personal knowledge may also give valuable insight.

A fundamental assumption for using this methodology is that similar type projects evolve similarly. If a different type of project was compared to this database, the manager would have to decide whether the baselines were applicable. Depending on the type of differences, the established baselines may or may not be of any value.

EXAMPLE 1:

Forty percent into coding a software manager finds that the lines of source code per software change is higher than normal. A list previously developed is examined to determine what the relative measure might indicate. The possible interpretations for a large number of lines of source code per software change might be:

- good code
- easily developed code
- influx of transported code
- near build or milestone date
- computer problems
- poor testing approach

If this were the only flagged measure the manager would then investigate each of the possibilities. If the value for the measure is close to the norm less concern is needed than if the value is further away.

If in addition to lines of source code per software change the number of computer runs per software change was higher than normal, the manager would also examine this measure. The possible interpretations for a large number of computer runs per software change might be:

- good code
- lots of testing
- change backlog
- poor testing approach

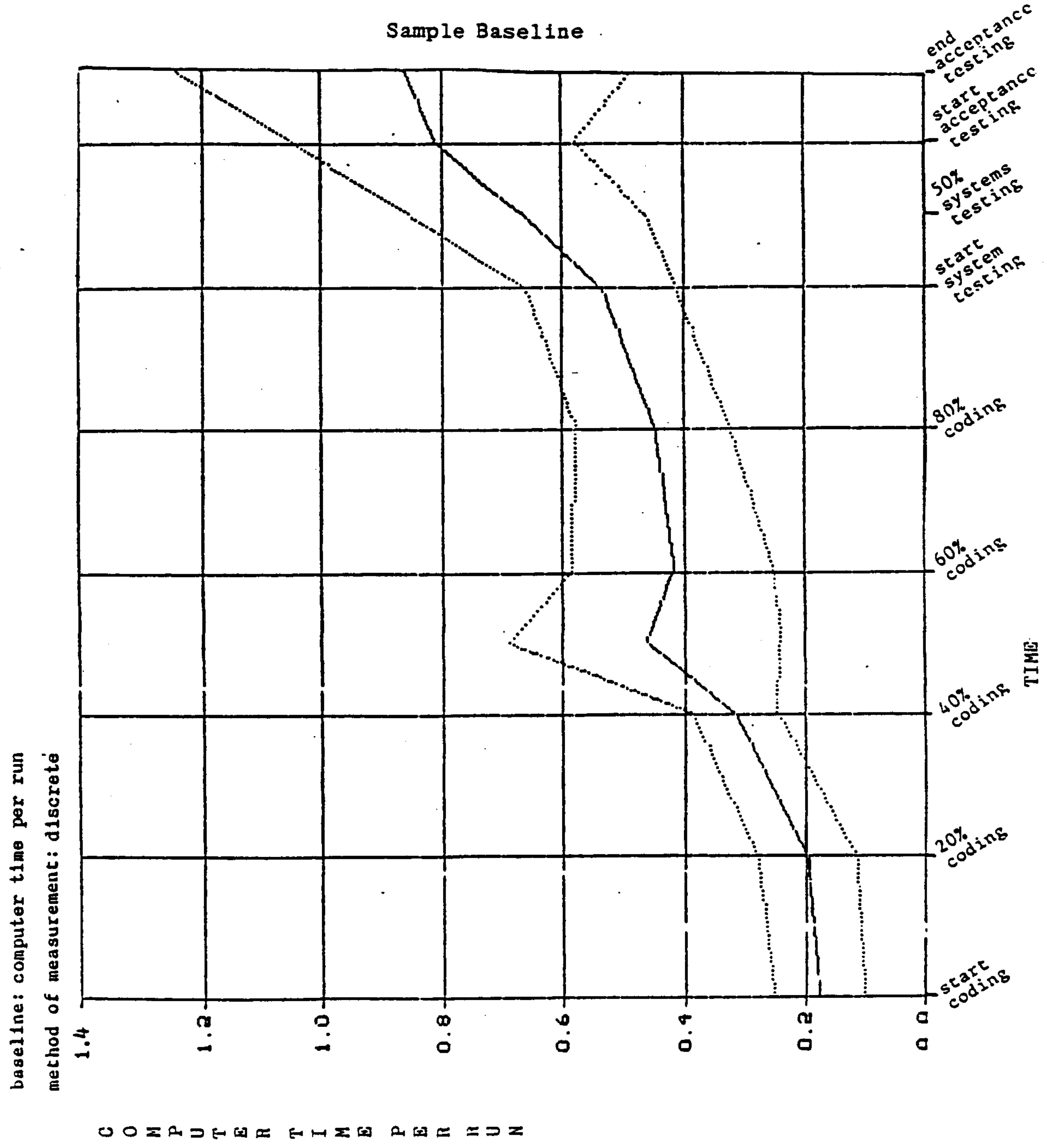
The union of the possible interpretations of these two measures indicates that the strongest possible interpretations are 1) good code and 2) a poor testing approach. The number of possibilities to investigate is smaller because these are the only measures which overlap. The manager must now examine the testing plan and decide whether either of these interpretations reflect what is actually occurring in the project. If these two possible interpretations do not reflect what is happening on the project, the manager would then examine the other interpretations.

III. Baseline Development

To develop a baseline one must first have variables whose measurements were taken weekly for several projects. Five variables in the SEL database were used. The lines of source code, number of software changes, and number of computer runs were collected on the growth history form. The amount of computer time and programmer hours were collected on the resource summary form. Measurement of these variables started near the beginning of coding. In this study, nine separate projects were examined whose development was documented, with sufficient data, in the SEL database. The projects ranged in size from 51-112K lines of source code with an average of 75K. No examination was done for the requirements or design phases.

Once the variables were chosen the average and standard deviation was computed for each baseline. Some baselines suffered from limited data points during the beginning of the coding phase. A couple of the projects, in which problems were known to have existed, were flagged as soon as data on these projects appeared, but this was fifty percent of the way into coding. It is not known how much earlier they would have appeared, if data

Sample Baseline



existed at the early intervals.

IV. Interpretation of Relative Measures

Once a set of baselines are established new projects may be compared to them and potential problems flagged. To interpret these flagged relative measures a list should be developed with each measures possible interpretations. Each list must consider the possible interpretations of the relative measure when it is either above normal or below normal. What each component variable actually measures should also be considered when the different lists are developed.

A list was developed with possible interpretations for each relative measure being examined in the context of the SEL environment. In another environment the interpretation of these measures might be different. These lists are subdivided into two categories; above and below normal. The above normal category contains possible interpretations for the relative measure when it is outside one standard deviation from the average in the positive direction. The below normal category refers to interpretations when the measure is outside one standard deviation from the mean in the negative direction.

One of the reasons this methodology works is because of the implicit interdependencies between different relative measures. To show these interdependencies more explicitly a cross reference chart has also been provided for each interpretation to indicate

Relative Measures Examined:

List 1 - Computer Runs per Line of Source Code
List 2 - Computer Time per Line of Source Code
List 3 - Software Changes per Line of Source Code
List 4 - Programmer Hours per Line of Source Code
List 5 - Computer Time per Computer Run
List 6 - Software Changes per Computer Run
List 7 - Programmer Hours per Computer Run
List 8 - Computer Time per Software Change
List 9 - Programmer Hours per Software Change

List 1: Computer Runs per Line of Source Code

type	interpretation	cross reference above normal	cross reference below normal
above normal			
	-low productivity	2 4	
	-high complexity	2 4 7 8 9	
	-lots of testing	2	6 7
	-removal of code (testing or transported)	2 3 4	
	-bad specifications	2 3 4	
below normal			
	-influx of transported code		2 3 4
	-near build or milestone date	6	2 3 4 8 9
	-little on line testing being done		2
	-little executable code being developed		2
	-computer problems		3

List 2: Computer Time per Line of Source Code

type	interpretation	cross reference above normal	cross reference below normal
above normal			
	-high complexity	1 4 7 8 9	
	-low productivity	1 4	
	-bad specifications	1 3 4	
	-lots of testing	1	6 7
	-unit testing being done	8	5
	-code being removed (testing or transported)	1 3 4	
below normal			
	-influx of transported code		1 3 4
	-near build or milestone date	6	1 3 4 8 9
	-little on line testing being done		1
	-code error prone	3 4 5 6	7 8 9
	-little executable code being written		1

List 3: Software Changes per Line of Source Code

type	interpretation	cross reference above normal	cross reference below normal
above normal			
	-good testing	6	8 9
	-error prone code	4 5 6	2 7 8 9
	-bad specifications	1 2 4	
	-code being removed (testing or transported)	1 2 4	
below normal			
	-influx of transported code		1 2 4
	-near build or milestone date	6	1 2 4 7 8
	-good code	8 9	6
	-poor testing program	8 9	6
	-change backlog		6
	-low complexity		4
	-computer problems		1

List 4: Programmer Hours per Line of Source Code

type	interpretation	cross reference	
		above normal	below normal
above normal			
	-high complexity	1 2 7 8 9	
	-error prone code	3 5 6	2 7 8 9
	-bad specifications	1 2 3	
	-code being removed (testing or transported)	1 2 3	
	-changes hard to isolate	7 8 9	
	-changes hard to make	7 9	
	-low productivity	1 2	
below normal			
	-influx of transported code		1 2 3
	-near build or milestone date	6	1 2 3 8 9
	-low complexity		3

List 5: Computer Time per Computer Run

type	interpretation	cross reference	
		above normal	below normal
above normal			
	-system & integration testing started early	6	
	-error prone code	3 4 6	2 7 8 9
	-compute bound algorithms being tested	8	
below normal			
	-unit testing going on	2 8	
	-easy errors being found		7 9

List 6: Software Changes per Computer Run

type	interpretation	cross reference	
		above normal	below normal
above normal			
	-good testing	3	8 9
	-system & integration testing started early	5	
	-error prone code	3 4 5	2 7 8 9
	-near build or milestone date		1 2 3 4 8 9
below normal			
	-good code	3 8 9	
	-lots of testing	1 2	7
	-poor testing program	3 8 9	
	-change backlog	3	

List 7: Programmer Hours per Computer Run

type	interpretation	cross reference	
		above normal	below normal
above normal			
	-high complexity	1 2 4 8 9	
	-modifications being made to recently transported code		9
	-changes hard to isolate	4 8 9	
	-changes hard to make	4 9	
below normal			
	-easy errors being fixed		5 9
	-error prone code	3 4 5 6	2 8 9
	-lots of testing	1 2	6

List 8: Computer Time per Software Change

type	interpretation	cross reference above normal	cross reference below normal
above normal			
	-good code	3 9	6
	-poor testing program	3 9	6
	-high complexity	1 2 4 7 9	
	-changes hard to isolate	4 7 9	
	-unit testing	2	5
	-compute bound algorithms being tested	5	
below normal			
	-near build or milestone date	6	1 2 3 4 9
	-good testing	3 6	9
	-error prone code	3 4 5 6	2 7 9

List 9: Programmer Hours per Software Change

type	interpretation	cross reference above normal	cross reference below normal
above normal			
	-good code	3 8	6
	-poor testing program	3 8	6
	-changes hard to isolate	4 7 8	
	-changes hard to make	4 7	
below normal			
	-good testing	3 6	8
	-near build or milestone date	6	1 2 3 4 8
	-easy changes		5 7
	-transported code being modified	7	
	-error prone code	3 4 5 6	2 7 8

other relative measures that can have the same interpretation. A number in the cross reference section indicates the list number of a relative measure that can have the same interpretation. The position of the list number in the 4-quadrant cross reference section indicates whether both interpretations are found with above normal values, both with below normal values, or one with above and the other with below normal values.

With these lists a set of flagged relative measures may be evaluated. When a relative measure is flagged, its associated list is examined for possible interpretations. Overlaps of this list with the lists of other flagged relative measures form the new list of what these relative measures together might indicate. The more overlaps a particular interpretation has, the greater the chance it is the correct interpretation. Interpretations with the same number of overlaps must be considered equally. The more relative measures flagged the more serious the problem may be. It is up to the manager to determine whether the deviation is good or bad.

V. Monitoring a Software Project's Development

Once the baselines have been developed and the lists of possible interpretations have been put together a software manager may monitor the actual development of a project. Example 1 demonstrated how a single interval may be interpreted. The following discussion will trace the development of an actual project. During the actual use of this methodology, influence would be exerted to correct problems as soon as they are identified. With this study, we must be content to study a projects evolution, without hindrance, and see at what points problems could of been detected.

Project twenty* was chosen for this examination because data existed throughout the projects development. In most respects project twenty was an average project. The project did have a lower than normal productivity rate. The lower rate may be partially explained by the fact the management was less experienced when compared to other projects. The project also suffered from some delayed staffing. Changes in staffing will be noted when the different time intervals are discussed.

The tables on the following page show which relative measures were flagged when project twenty was compared to the baselines for each stage of development. The numerical values represent how many standard deviations each flagged relative measure was from the baseline. The baseline for each relative measure was calculated using all nine projects.

Start of Coding:

At the start of coding only one relative measure is flagged. The smaller than normal number of software changes per line of source code using the discrete approach reflects work done during

* The numbering convention used is an extension of the one first used by Bailey and Basili [Bailey81].

project: 20

method of measurement: cumulative

number of standard deviations from norm										relative measures
start code	20% code	40% code	50% code	60% code	80% code	start sys	50% sys	start accpt	end	
									1.3	>1 SD programmer hours/lines of source
						1.1				>1 SD runs/lines of source
						1.8	1.5	1.2		>1 SD computer time/lines of source
		1.1	1.2	1.1		1.1				<1 SD programmer hours/run

method of measurement: discrete

number of standard deviations from norm										relative measures
start code	20% code	40% code	50% code	60% code	80% code	start sys	50% sys	start accpt	end	
	1.0	1.1	1.8			1.5		2.0	2.4	>1 SD programmer hours/lines of source
	1.2		1.8			1.8		1.7		>1 SD runs/lines of source
1.1										<1 SD changes/lines of source
	1.1		1.1			2.0		2.0	2.4	>1 SD changes/lines of source
	1.2	1.3	1.7			2.1		2.0		>1 SD computer time/lines of source
	1.2									<1 SD programmer hours/run
						1.2				>1 SD computer time/change

the design phase. The lists designed in the previous section were directed towards code production and testing and do not apply to this time interval when using the discrete approach. This measure may indicate good specifications or lots of PDL being generated. The manager might want to examine this measure later if it constantly repeated. Since it is the only measure flagged at this time it will be ignored.

20% Coding:

The flagged relative measures found using the discrete approach at this point represent the work done from the start of coding until twenty percent of the way through coding. The list of possible interpretations for the flagged relative measures, generated from the lists made previously for the individual relative measure, would look like:

# overlaps	interpretation
3	bad specifications
3	code removed
2	low productivity
2	high complexity
2	error prone code
1	lots of testing
1	good testing
	changes hard to isolate
	changes hard to make
	unit testing being done
	easy errors being found

The strongest interpretations are bad specifications and code being removed. If the actual history is examined one finds that during this period there were a lot of specifications being changed. This resulted in code which was to be modified being discarded and new code being written. During the early period lots of PDL was being produced but very little new executable code. The list of possible interpretations does show that low productivity is also a strong possibility.

40% Coding:

The flagged relative measures which appear using the cumulative approach, from this time period on, are stronger indicators than the ones used in the first couple of intervals because the average is computed using more data points. The use of the discrete approach for the interval of twenty to forty percent is still dependent on three data points. The list of possible interpretations for this time period is:

# overlaps	interpretation
1	low productivity
1	high complexity
1	error prone code
1	bad specifications
1	code being removed
	changes hard to isolate
	changes hard to make
	lots of testing
	unit testing being done
	good testing
	easy errors

The number of possibilities is larger with this set of possible interpretations. Five interpretations are slightly stronger than the others. During the actual development, the first release of the project was made. The amount of code actually written was also lower than normal during this period. The use of the discrete approach gives a stronger feeling that code is not being written. Transported code tends to be installed in large blocks which can be isolated using the discrete approach.

50% Coding:

The relative measures flagged during this period are the same as the ones flagged at the twenty percent coding interval. The deviation from the norm for this interval is larger. The larger deviation may indicate a more serious problem. The problem may of been just as serious earlier but without the extra data points, that are now available, it could not be determined. The possible interpretations may be taken from the list developed earlier. Bad specifications and code removal were not factors during this period. The next three highest priority interpretations were; high complexity, error prone code, and low productivity. In addition to this the manager should be concerned with the continued appearance of the relative measure, programmer hours per computer run, as seen using the cumulative approach. This may indicate a lot of testing going on. This in conjunction with error prone code as a possible interpretation may indicate trouble. During actual development this period was spent developing code for the second release. The project manager felt that code was still not being developed quickly enough during this period.

60% Coding:

Only one relative measure is shown at this interval. The number of programmer hours per computer run using the cumulative approach is lower than normal for the third consecutive time. This should concern the manager because when examining the list for this measure one finds:

error prone code
lots of testing
easy errors being fixed

Since the occurrence of this measure is persistent it may indicate that the problem was corrected but not enough effort was expended to completely compensate for the past problems. It might also indicate the problem still exists. During the actual project it was found that while a lot of code was written, it had not been throughly tested. Release two was made during this period which could explain a heavy test load. Two additional staff members were added to the project during this phase to aid in coding and testing.

80% Coding:

The eighty percent coding interval does not show any measures outside the normal bounds. The addition of two staff members during the sixty percent coding phase, as well as the addition of a senior staff member during this phase, appears to have adjusted the project back along the lines of normal development. To fully compensate for the earlier problems one might expect some of the measures to swing in the other direction away from the average. The fact this over correction did not occur might explain the problems encountered in the next section.

Start of System and Integration Testing:

The flagged relative measures at this time period reflect the build up of effort for the third and final release. The list of possible interpretations for the collective set of flagged measures looks like:

# overlaps	interpretation
3	high complexity
3	bad specifications
3	code being removed
2	error prone code
2	low productivity
2	lots of testing
1	changes hard to isolate
1	unit testing being done
1	good code
1	poor testing
	changes hard to make
	good testing
	compute bound algorithms being run
	easy errors being fixed

Since the code did have a past history of poor testing an unusually large build up of testing should be expected. The two interpretations that apply most to this situation are lots of testing and error prone code.

50% System and Integration Testing:

Only one relative measure is flagged at this interval. This measure was flagged using the cumulative approach. An examination of the measure at the previous interval shows a very high value. A slow drop off from this high measure is to be expected when using the cumulative approach. An examination of possible interpretations that would apply for this period of development include:

high complexity
lots of testing
unit testing being done
testing code being removed

A lot of testing is certainly indicated by past history.

Start Acceptance Testing:

The relative measures flagged at this interval reflects the build up in testing before the start of acceptance testing. The list of possible interpretations looks like:

# overlaps	interpretation
3	bad specifications
3	code being removed
2	high complexity
2	low productivity
1	error prone code
1	lots of testing
	changes hard to isolate
	changes hard to make
	unit testing being done
	good testing

Since little code was being developed during the testing period, a large amount of testing with errors being found is the most reasonable interpretation of these flagged measures. The early history of poor testing may be seen here with errors being uncovered late.

End Acceptance Testing:

The two flagged relative measures at the end of acceptance testing reflect the clean up effort being made on the code. An average amount of computer time and an average number of computer runs indicates that the acceptance testing is going well. The project was behind schedule due to the earlier problems encountered. Clean up was done during the acceptance testing phase in an attempt to get the project out the door as soon as possible.

As seen in this example, the problems that occur during a projects development are reflected in the values calculated for the relative measures. The methodology preposed can be used to monitor projects. The number of possible interpretations increases with each new flagged relative measure. The ordering of the measures by the number of overlaps provides an easy method of sorting the possible interpretations by priority. Another method of sorting the possible interpretations could include a factor that considers both the number of overlaps and the probability of a given interpretation being the cause at a given interval. The weighting of interpretations for a given interval could be calculated using the pattern of occurrence of the different interpretations which have appeared during the same interval in past projects.

VI. An Alternate Approach

Flagged relative measures might also be interpreted using a decision support system. The data for the various relative measures would be stored in a knowledge base along with a set of production rules. To evaluate a project the values for each relative measure would be entered into the system. The knowledge base would compare the relative measures to their respective baselines, determine which relative measures were outside the norm, and interpret these relative measures using the production rules. A list of possible interpretations ordered by probability would be generated as a result.

The difference between a decision support system and the approach presented in this paper is the method of interpreting the flagged relative measures. Each production rule in the decision support system is the logical disjunction of several flagged measures which yields a given interpretation. Each production rule is assigned a confidence rating which is then used to rate the possible interpretations. The lists for the relative measures provided earlier in the paper may be easily converted to production rules using the cross reference section. To develop the production rules for an interpretation one must generate the various combinations of relative measures which might reasonably imply the interpretation. Some relative measures may not imply a particular interpretation unless they are found in conjunction with another relative measure. Once the production rules are known and a knowledge base constructed a decision support system may be built. For an example of a domain independent decision support system see Reggia and Perricone [Reggia82].

VII. Summary

The methodology presented in this paper showed that invariant relationships exist for similar projects. New projects may be compared to the baselines of these invariant relationships to determine when projects are getting off track.

The ability of the manager to interpret the measures that fall outside the norm is dependent on the amount of information the underlying variables convey. The manager must decide what attributes are to be measured (e.g. productivity) and pick variables that are closely related to them and are also measurable throughout the project. As an example, a variable like lines of code may be too general when measuring productivity. Measuring the newly developed code, either source code or executable code, would be more informative since these variables are more directly related to effort. How applicable an interpretation is for the period currently being examined should also be considered when ordering the list. The variables the manager finally decides on are then combined to form relative measures.

One method of interpreting a relative measure is by associating lists of possible interpretations with it. When a relative measure appears outside the norm, the list of possible interpretations is considered. If more than one relative measure is outside the norm the lists are combined. The more times a possible interpretation is repeated in the lists, the greater the probability it is the cause. How applicable an interpretation is for the period being examined should also be considered when ordering the list. The manager must investigate the suggested causes to determine the real one.

VIII. Conclusion

The ability to monitor a projects development and detect problems as they develop may be feasible. The methodology proposed showed favorable results when examining a past case.

The use of baselines and lists of interpretations for comparing projects provides an easy method for monitoring software development. Both the baselines and the lists of interpretations may be updated as new projects are developed. As more knowledge is gleaned the accuracy of this system should improve and provide a valuable tool for the manager.

Bibliography

[Bailey81]
Bailey, John W. and Victor R. Basili, A Meta-Model for Software Development Resource Expenditures, Proceedings, Fifth International Conference on Software Engineering, September 1981.

[Basili81]
Basili, Victor R. and Karl Freburger, Programming Measurement and Estimation in the Software Engineering Laboratory, Journal of Systems and Software, 1981.

[Card82]
Card, David, Frank McGarry, Jerry Page, Suellen Eslinger, and Victor Basili, The Software Engineering Laboratory, SEL-81-104, Software Engineering Laboratory Series, Goddard Space Flight Center, February 1982.

[Church82]
Church, Victor, David Card, Frank McGarry, Jerry Page, and Victor Basili, Guide To Data Collection, SEL-81-101, Software Engineering Laboratory Series, Goddard Space Flight Center, August 1982.

[Manley82]
The Role of Measurements in Programming Technology, Lecture presented at University of Maryland, November 15, 1982.

[Minsky75]
Minsky, M. L., A Framework for the Representation of Knowledge, The Psychology of Computer Vision, pp. 211-280, McGraw Hill, New York, 1975.

[Reggia82]
Reggia, James and Barry Perricone, KMS Manual, TR-1136, Department of Mathematics, University of Maryland Baltimore County, January 1982.

[SEL82]
SEL,, Collected Software Engineering Papers: Volume 1, SEL-82-004, Software Engineering Laboratory Series, Goddard Space Flight Center, July 1982.

[Walston77]
Walston, C. E. and C. P. Felix, A Method of Programming Measurement and Estimation, IBM Systems Journal, January 1977.

D7-61

N87-24902

Technical Report TR-1195

August 1982
NSG-5123

SOFTWARE ERRORS AND COMPLEXITY: AN EMPIRICAL INVESTIGATION*

Victor R. Basili and Barry T. Perricone

N 91576

*Research supported in part by National Aeronautics and Space Administration grant NSG-5123 to the University of Maryland.. Computer time supported in part through the facilities of the Computer Science Center of the University of Maryland.

PRECEDING PAGE BLANK NOT FILMED

PAGE 4-72 INTENTIONALLY BLANK

SOFTWARE ERRORS AND COMPLEXITY:
AN EMPIRICAL INVESTIGATION

Victor R. Basili and Barry T. Perricone

Department of Computer Science

University of Maryland

College Park, Md.

1982

ABSTRACT

The distributions and relationships derived from the change data collected during the development of a medium scale satellite software project shows that meaningful results can be obtained which allow an insight into software traits and the environment in which it is developed. Modified and new modules were shown to behave similarly. An abstract classification scheme for errors which allows a better understanding of the overall traits of a software project is also shown. Finally, various size and complexity metrics are examined with respect to errors detected within the software yielding some interesting results.

1.0 INTRODUCTION

The discovery and validation of fundamental relationships between the development of computer software, the environment in which the software is developed, and the frequency and distribution of errors associated with the software are topics of primary concern to investigators in the field of software engineering. Knowledge of such relationships can be used to provide an insight into the characteristics of computer software and the effects that a programming environment can have on the software product. In addition, it can provide a means to improve the understanding of the terms reliability and quality with respect to computer software. In an effort to acquire a knowledge of these basic relationships, change data for a medium scale software project was analyzed (e.g., change data is any documentation which reports an alteration made to the software for a particular reason).

In general, the overall objectives of this paper are threefold : first, to report the results of the analyses; second, to review the results in the context of those reported by other researchers; and third, to draw some conclusions based on the aforementioned. The analyses presented in this paper encompass various types of distributions based on the collected change data. The most important of which are the error distributions observed within the software project.

In order for the reader to view the results reported in this paper properly, it is important that the terms used throughout this paper and the environment in which the data was collected are clearly defined. This is pertinent since many of the terms used within this paper have appeared in the general literature often to denote different concepts. Understanding the environment will allow the partitioning of the results into two classes: those which are dependent on and those which are independent of a particular programming environment.

1.1 DESCRIPTION OF THE ENVIRONMENT

The software analyzed within this paper is one of a large set of projects being analyzed in the Software Engineering Laboratory (SEL). The particular project analyzed in this paper is a general purpose program for satellite planning studies. These studies include among others: mission maneuver planning; mission lifetime; mission launch; and mission control. The overall size of the software project was approximately 90,000 source lines of code. The majority of the software project was coded in FORTRAN. The system was developed and executes on an IBM 360.

The developers of the analyzed software had extensive experience with ground support software for satellites. The analyzed system represents a new application for the development group, although it shares many similar algorithms with the system studied here.

It is also true that the requirements for the system analyzed kept growing and changing, much more so than for the typical ground support software normally built. Due to the commonality of algorithms from existing systems, the developers re-used the design and code for many algorithms needed in the new system. Hence a large number of re-used (modified)
modules became part of the new system analyzed here.

An approximation of the analyzed software's life cycle is displayed in Figure 1 . This figure only illustrates the approximate duration in time of the various phases of the software's life cycle. The information relating the amount of manpower involved with each of the phases shown was not specific enough to yield meaningful results, so it was not included.

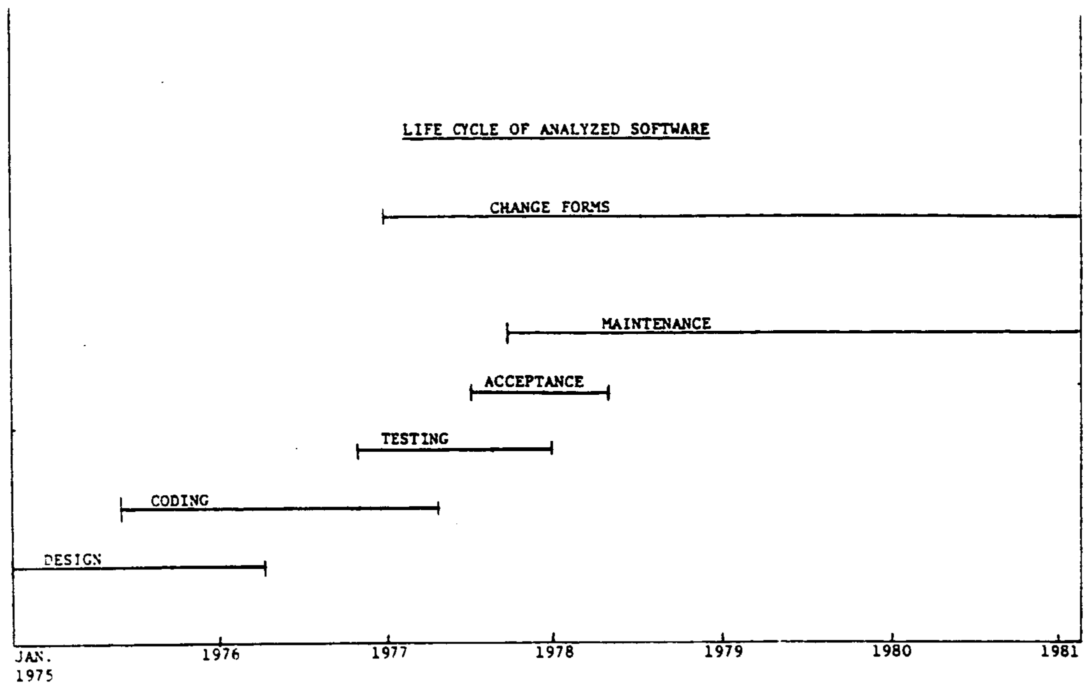


Figure 1

1.2 TERMS

This section presents the definitions and associated contexts for the terms used within this paper. A discussion of the concepts involved with these terms is also given when appropriate.

Module: A module is defined as a named subfunction, subroutine, or the main program of the software system. This definition is used since only segments written in FORTRAN which contained executable code were used for the analyses. Change data from the segments which constituted the data blocks, assembly segments, common segments, or utility routines were not included. However, a general overview of the data available on these types of segments is presented in Section 4.0 for completeness.

There are two types of modules referred to within this paper. The first type is denoted as modified. These are modules which were developed for previous software projects and then modified to meet the requirements of the new project. The second type is referred to as new. These are modules which were developed specifically for the software project under analyses.

The entire software project contained a total of 517 code segments. This quantity is comprised of 36 assembly segments, 370 FORTRAN segments, and 111 segments that were either common modules, block data, or utility routines. The number of code segments which met the adopted module definition was 370 out of 517 which is 72% of the total modules and constitutes the majority of the software project. Of the modules found to contain errors 49% were categorized as modified and 51% as new modules.

Number of Source and Executable Lines: The number of source lines within a module refers to the number of lines of executable code and comment lines contained within it. The number of executable lines within a module refers to the number of executable statements, comment lines are not included.

Some of the relationships presented in this paper are based on a grouping of modules by module size in increments of 50 lines. This means that a module containing 50 lines of code or less was placed in the module size of 50; modules between 51 and 100 lines of code into the module size of 100, etc. The number of modules which were contained in each module size is given in Table 1 for all modules and for modules which contained errors (i.e., a subset of all modules) with respect to source and executable lines of code.

	Number modules			
	All Modules		Modules with Errors	
Number of Lines	Source	Exececutable	Source	Executable
0-50	53	258	3	49
51-100	107	70	16	25
101-150	80	26	20	13
151-200	56	13	19	7
201-250	34	1	12	1
251-300	14	1	9	0
301-350	7	1	4	1
351-400	9	0	7	0
>400	10	0	6	0
Total	370	370	96	96

Table 1

Error: Something detected within the executable code which caused the module in which it occurred to perform incorrectly (i.e., contrary to its expected function).

Errors were quantified from two view points in this paper, depending upon the goals of the analysis of the error data. The first quantification was based on a textual rather than a conceptual viewpoint. This type of error quantification is best illustrated by an example. If a "*" was incorrectly used in place of a "+" then all occurrences of the "*" will be considered an error. This is the situation even if the "*"´s appear on the same line of code or within multiple modules. The total number of errors detected in the 370 software modules analyzed was 215 contained within a total of 96 modules, implying 26% of the modules analyzed contained errors.

The second type of quantification was used to measure the effect of an error across modules, textual errors associated with the same conceptual problem were combined to yield one conceptual error. Thus in the example above, all incorrectly used *´s replaced by +´s in the same formula were combined and the total number of modules effected by that error are listed. This is done only for the errors reported in Figure 2. There are a total of 155 conceptual errors. All other studies in this paper are based upoon the first type of quantification described.

Statistical Terms and Methods: All linear regressions of the data presented within this paper employed as a criterion of goodness the least squares principle (i.e., "choose as the 'best fitting' line that one which minimizes the sum of squares of the deviations of the observed values of y from those predicted" [1]).

Pearson's product moment coefficient of correlation was used as an index of the strength of the linear relationship independent of the respective scales of measurement for y and x. This index is denoted by the symbol r within this paper. The measure for the amount of variability in y accounted for by linear regression on x is denoted as r2 within this paper.

All of the equations and explanations for these statistics can be found in [1]. It should be noted that other types of curve fits were conducted on the data. The results of these fits will be mentioned later in the paper.

Now that the software's environment and the key terms used within the paper have been defined and outlined, a discussion of the basic quantification of the data collected, the relationships and distributions derived from this quantification, and the resulting conclusions are presented.

2.0 BASIC DATA

The change data analyzed was collected over a period of 33 months, August 1977 through May 1980. These dates correspond in time to the software phases of coding, testing, acceptance, and maintenance (Figure 1) . The data collected for the analyses is not complete since changes are still being made to the software analyzed. However, it is felt that enough data was viewed in order to make the conclusions drawn from the data significant.

The change data was entered on detailed report sheets which were completed by the programmer responsible for implementing the change. A sample of the change report form is given in the Appendix. In general, the form required that several short questions be answered by the programmer implementing the change. These queries allowed a means to document the cause of a change in addition to other characteristics and effects attributed to the change. The majority of this information was found useful in the analyses. The key information used in the study from the form was: the data of the change or error discovery, the description of the change or error, the number of components changed, the type of change or error, and the effort needed to correct the error.

It should be mentioned that the particular change report form shown in the Appendix was the most current form but was not uniformly used over the entire period of this study. In actuality there were three different versions of the change report form, not all of which required the same set of questions to be answered. Therefore , for the data that was not present on one type of form but could be inferred, the inferred value was used. An example of such an inference would be that of determining the error type. Since the error description was given on all of the forms the error type could be inferred with a reasonable degree of reliability. Data not incorporated into a particular data set used for an analysis was that data for which this inference was deemed unreliable. Therefore, the reader should be alert to the cardinality of the data set used as a basis for some of the relationships presented in this paper. There was a total of 231 change report forms examined for the purpose of this paper.

The consistency and partial validity of the forms was checked in the following manner. First, the supervisor of the project looked over the change report forms and verified them (denoted by his or her signature and the date). Second, when the data was being reduced for analysis it was closely examined for contradictions. It should be noted that interviews with the individuals who filled out the change forms were not conducted. This was the major difference between this work and other error studies performed by the Software Engineering Laboratory, where interviews were held with the programmers to help clarify questionable data (8).

The review of the change data as described above yielded an interesting result. The errors due to previous miscorrections showed to be three times as common after the form review process was performed, i.e. before the review process they accounted for 2% of the errors and after the review process they accounted for 6% of the errors. These recording errors are probably attributable to the fact that the corrector of an error did not know the cause was due to a previous fix because the fix occurred several months earlier or was made by a different programmer, etc.

3.0 RELATIONSHIPS DERIVED FROM DATA

This section presents and discusses relationships derived from the change data.

3.1 CHANGE DISTRIBUTION BY TYPE

Types of changes to the software can be categorized as error corrections or modifications (specification changes, planned enhancements, clarity and optimization improvements). For this project, error corrections accounted for 62% of the changes and modifications 38%. In studies of other SEL projects, errors corrections ranged from 40% to 64% of the changes.

3.2 ERROR DISTRIBUTION BY MODULES

Figure 2 shows the effects of an error in terms of the number of modules that had to be changed. (Note that these errors here are counted as conceptual errors.) It was found that 89% of the errors could be corrected by changing only one module. This is a good argument for the modularity of the software. It also shows that there is not a large amount of interdependence among the modules with respect to an error.

NUMBER OF MODULES AFFECTED BY AN ERROR (data set: 211 textual errors)
174 conceptual errrors)

#ERRORS	#MODULES AFFECTED
155 (89%)	1
9	2
3	3
6	4
1	5

Figure 2

Figure 3 shows the number of errors found per module. The type of module is shown in addition to the overall total number of modules found to contain errors.

NUMBER OF ERRORS PER MODULE (data set: 215 errors)

#MODULES	NEW	MODIFIED	#ERRORS/MODULE
36	17	19	1
26	13	13	2
16	10	6	3
13	7	6	4
4	1**	3*	5
1	1**		7

Figure 3

The largest number of errors found were 7 (located in a single new module) and 5 (located in 3 different modified modules and 1 new module). The remainder of the errors were distributed almost equally among the two types of modules.

The effort associated with correcting an error is specified on the form as being (1) 1 hour or less, (2) 1 hour to 1 day, (3) 1 day to 3 days, (4) more than 3 days. These categories were chosen because it was too difficult to collect effort data to a finer granularity. To estimate the effort for any particular error correction, an average time was used for each category, i.e. assuming an 8 hour day, an error correction in category (1) was assumed to take .5 hours, an error correction in category (2) was assumed to take 4.5 hours, category (3) 16 hours, and category (4) 32 hours.

The types of errors found in the three most error prone modified modules (* in Figure 3) and the effort needed to correct them is shown in Table 2. If any type contained error corrections from more than one error correction category, the associated effort for them was averaged. The fact that the majority of the errors detected in a module was between one and three shows that the total number of errors that occurred per module was on the average very small.

The twelve errors contained in the two most error prone new modules (** in Figure 3) are shown in Table 3 along with the effort needed to correct them.

	NUMBER OF ERRORS (15 total)	AVERAGE EFFORT[TO CORRECT
misunderstood or incorrect specifications	8	24 hours
incorrect design or implementation of a module component	5	16 hours
clerical error	2	4.5 hours

EFFORT TO CORRECT ERRORS IN THREE MOST ERROR PRONE MODIFIED MODULES
Table 2

	NUMBER OF ERRORS (12 total)	AVERAGE EFFORT TO CORRECT
misunderstood or incorrect requirements	8	32 hours
incorrect design or implementation of a module	3	0.5 hours
clerical error	1	0.5 hours

EFFORT TO CORRECT ERRORS IN THE TWO MOST ERROR PRONE NEW MODULES
Table 3

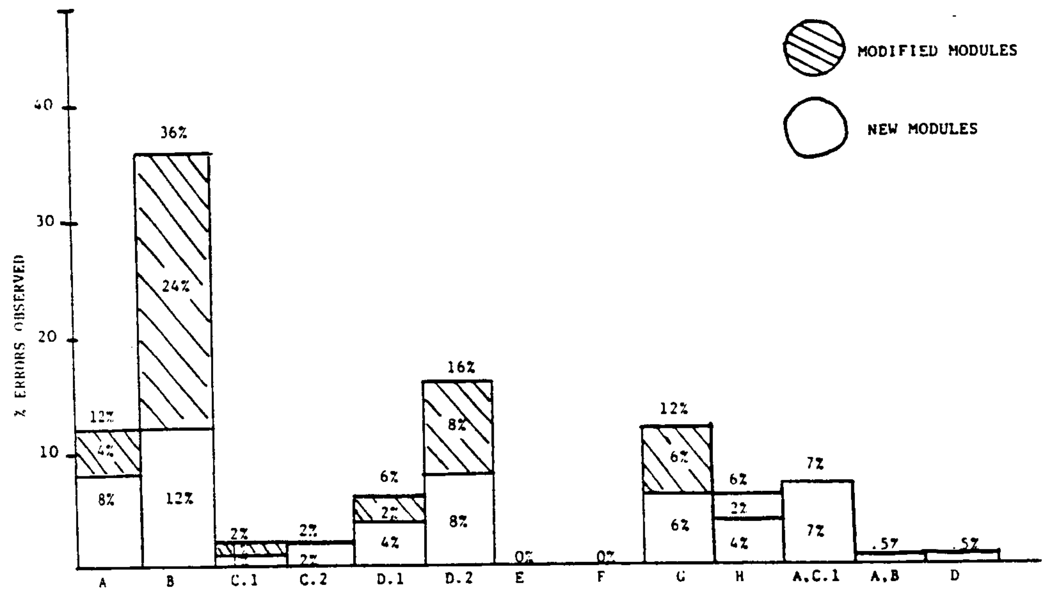
3.3 ERROR DISTRIBUTION BY TYPE

In Figure 4 the distribution of errors are shown by type. It can be seen that 48% of the errors were attributed to incorrect or misinterpreted functional specifications or requirements.

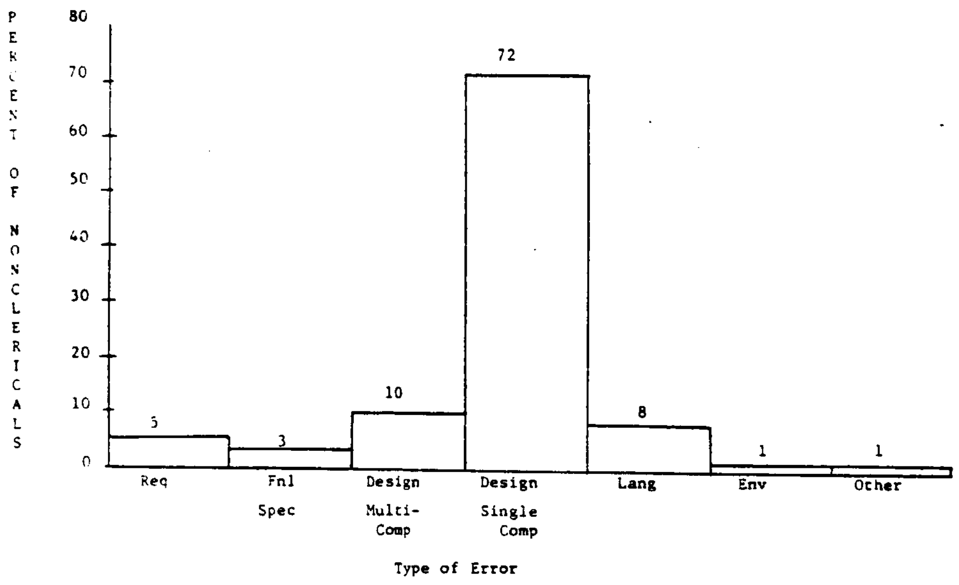
The classification for error used throughout the Software Engineering Laboratory is given below. The person identifying the error indicates the class for each error.

A: Requirements incorrect or misinterpreted
B: Functional specification incorrect or misinterpreted
C: Design error invloving several components
  1. mistaken assumption about value or structure of data
  2. mistake in control logic or computation of an expression
D: Error in design or implementation of single component
  1. mistaken assumption about value or structure of data
  2. mistake in control logic or computation of an expression
E: Misunderstanding of external environment
F: Error in the use of programming language/compiler
G: Clerical error
H: Error due to previous miscorrection of an error

The distribution of these errors by source is plotted in Figure 4 with the appropriate subdistribution of new and modified errors displayed. This distribution shows the majority of errors were the result of the functional specification being incorrect or misinterpreted . Within this category, the majority of the errors (24%) involved modified modules This is most likely due to the fact that the modules reused were taken from another system with a different application. Thus, even though the basic algorithms were the same, the specification was not well enough defined or appropriately defined for the modules to be used under slightly different circumstances.



SOURCES OF ERRORS
Figure 4



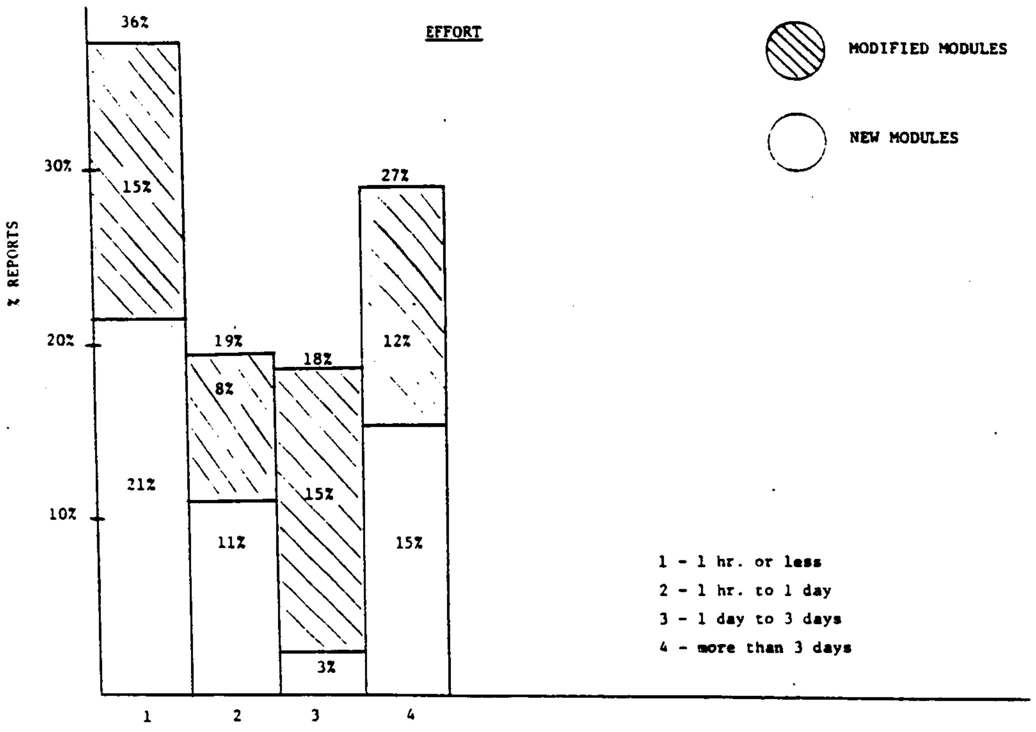
SOURCES OF ERROR ON OTHER PROJECTS
Figure 5

The distribution in Figure 4 should be compared with the distribution of another system developed by the same organization shown in Figure 5. Figure 5 represents a typical ground support software system and was rather typical of the error distributions for these systems. It is different from the distribution for the system we are discussing in this paper however, in that the majority of the errors were involved in the design of a single component. The reason for the difference is that in ground support systems, the design is well understood, the developers have had a reasonable amount of experience with the application. Any re-used design or code comes from similar systems, and the requirements tend to be more stable. An analysis of the two distributions makes the differences in the development environments clear in a quantitative way.

The percent of requirements and specification errors is consistent with the work of Endres´[1]. Endres found that 46% of the errors he viewed involved the misunderstanding of the functional specifications of a module. Our results are similar even though Endres´ analysis was based on data derived from a different software project and programming environment. The software project used in Endres´ analysis contained considerably more lines of code per module, was written in assembly code, and was within the problem area of operating systems. However, both of the software systems Endres analyzed did contain new and modified modules.

Of the errors due to the misunderstanding of a module´s specifications or requirements (48%), 20% involved new modules while 28% involved modified modules.

Although the existence of modified modules can shrink the cost of coding, the amount of effort needed to correct errors in modified modules might outweigh the savings. The effort graph (Figure 6) supports this viewpoint: 50% of the total effort required for error correction occurred in modified modules; errors requiring one day to more than three days to correct accounted for 45% of the total effort with 27% of this effort attributable to modified modules within these greater effort classes. Thus, errors occurring in new modules required less effort to correct than those occurring in modified modules.



EFFORT GRAPH
Figure 6

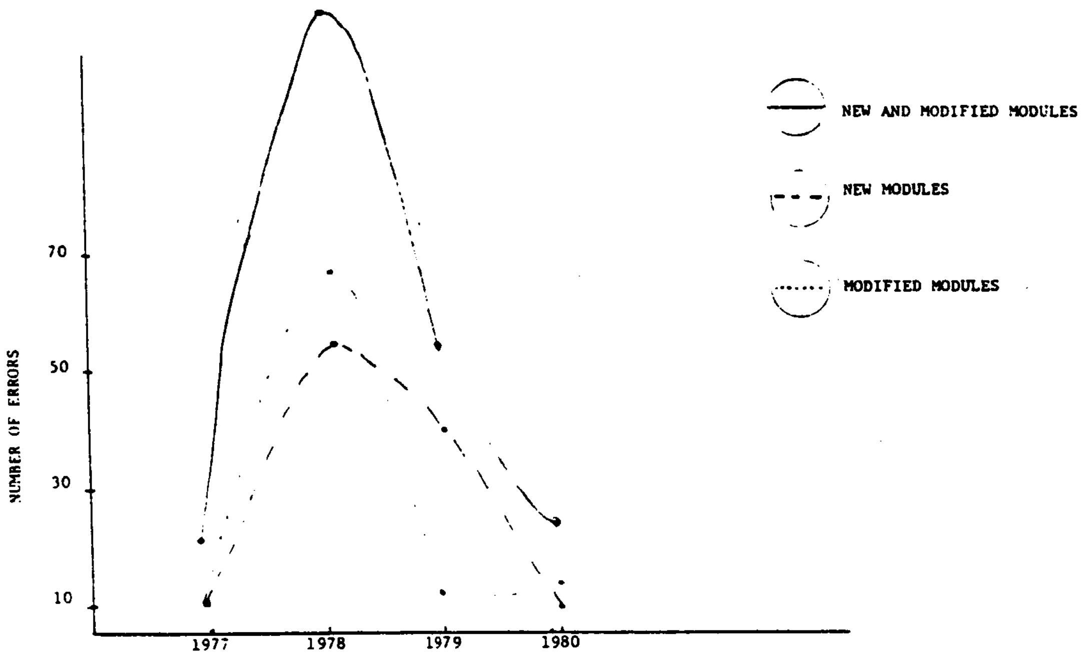
The similarity between Endres´ results and those reported here tend to support the statement that independent of the environment and possibly the module size, the majority of errors detected within software is due to an inadequate form or interpretation of the specifications. This seems especially true when the software contains modified modules.

In general, these observations tend to indicate that there are disadvantages in modifying a large number of already existing modules to meet new specifications. The alternative of developing a new module might be better in some cases if there does not exist good specifications for the existing modules.

3.4 OVERALL NUMBER OF ERRORS OBSERVED

Figure 7 displays the number of errors observed in both new and modified modules. The curve representing total modules (new and modified) is basically bell-shaped. One interpretation is that up to some point errors are detected at a relatively steady rate. At this point at least half of the total "detected-undetected" errors have been observed and the rate of discovery thereafter decreases. It may also imply the maintainers are not adding too many new errors as the system evolves.

It can be seen, however, that errors occurring in modified modules are detected earlier and at a slightly higher rate than those of new modules. One hypothesis for this is that the majority of the errors observed in modified modules are due to the misinterpretation of the functional specifications as was mentioned earlier in the paper. Errors of this type would certainly be more obvious since they are more blatant than those of other types and therefore, would be detected both earlier and more readily.(See next section.)



	1977	1978	1979	1980
NEW	10	54	40	9
MOD	10	67	11	14
COMB	20	121	51	23

NUMBER OF ERRORS OCCURRING IN MODULES
Figure 7

3.5 ABSTRACT ERROR TYPES

An abstract classification of errors was adopted by the authors which classified errors into one of five categories with respect to a module: (1) initialization; (2) control structure; (3) interface; (4) data; and (5) computation. This was done in order to see if there existed recurring classes of errors present in all modules independent of size. These error classes are only roughly defined so examples of these abstract error types are presented below. It should be noted that even though the authors were consistant with the categorization for this project, another error analyst may have interpreted the categories differently.

Failure to initialize or re-initialize a data structure properly upon a module's entry/exit would be considered an initialization error. Errors which caused an "incorrect-path" in a module to be taken were considered control errors. Such a control error might be a conditional statement causing control to be passed to an incorrect path. Interface errors were those which were associated with structures existing outside the module's local environment but which the module used. For example, the incorrect declaration of a COMMON segment or an incorrect subroutine call would be an interface error. An error in the declaration of the COMMON segment was considered an interface error and not an initialization error since the COMMON segment was used by the module but was not part of its' local environment. Data error would be those errors which are a result of the incorrect use of a data structure. Examples of data errors would be the use of incorrect subscripts for an array, the use of the wrong variable in an equation, or the inclusion of an incorrect declaration of a variable local to the module. Computation errors were those which caused a computation to erroneously evaluate a variable's value. These errors could be equations which were incorrect not by virtue of the incorrect use of a data structure within the statement but rather by miscalculations. An example of this error might be the statement A = B + 1 when the statement really needed was A = B/C + 1.

These five abstract categories basically represent all activities present in any module. The five categories were further partitioned into errors of commission and omission. Errors of <u>commission</u> were those errors present as a result of an incorrect executable statement. For example, a commissioned computational error would be A = B * C where the '*' should have been '+'. In other words, the operator was present but was incorrect. Errors of <u>omission</u> were those errors which were a result of forgetting to include some entity within a module. For example, a computational omission error might be A = B when the statement should have read A = B + C. A parameter required for a subroutine call but not included in the actual call would be an example of an interface omission error. In both of the above examples some aspect needed for the correct execution of a module was forgotten.

The results of this abstract classification scheme as discussed above is given in Figure 8. Since there were approximately an equal amount of new (49) and modified (47) modules viewed in the analysis, the results do not need to be normalized. Some errors and thereby modules were counted more than once since it was not possible to associate some errors with a single abstract error type based on the error description given on the change report form.

	commission		omission	
	new	modified	new	modified
initialization	2	9	5	9
control	12	2	16	6
interface	23	31	27	6
data	10	17	1	3
computation	16	21	3	3
	28%	36%	23%	12%
	64%		35%	

	total			
	new	modified		
initialization	7	18	---	25 (11%)
control	28	8	---	36 (16%)
interface	50	37	---	87 (39%)
data	11	20	---	31 (14%)
computation	19	24	---	43 (19%)
	115	107		

ABSTRACT CLASSIFICATION OF ERRORS
Figure 8

According to Figure 8, interfaces appear to be the major problem regardless of the module type. Control is more of a problem in new modules than in modified modules. This is probably because the algorithms in the old modules had more test and debug time. On the other hand, initialization and data are more of a problem in modified modules. These facts, coupled with the small number of errors of omission in the modified modules might imply that the basic algorithms for the modified modules were correct but needed some adjustment with respect to data values and initialization for the application of that algorithm to the new environment.

3.6 MODULE SIZE AND ERROR OCCURRENCE

Scatter plots for executable lines per module versus the number of errors found in the module were plotted. It was difficult to see any trend within these plots so the number of errors/1000 executable lines within a module size was calculated (Table 4).

Module Size	Errors/1000 lines
50	16.0
100	12.6
150	12.4
200	7.6
>200	6.4

ERRORS/1000 EXECUTABLE LINES (INCLUDES ALL MODULES)
Table 4

The number of errors was normalized over 1000 executable lines of code in order to determine if the number of detected errors within a module was dependent on module size. All modules within the software were included, even those with no errors detected. If the number of errors/1000 exececutable lines was found to be constant over module size this would show independence. An unexpected trend was observed: Table 4 implies that there is a higher error rate within smaller sized modules. Since only the executable lines of code were considered the larger modules were not COMMON data files. Also the larger modules will be shown to be more complex than smaller modules in the next section. Then how could this type of result occur?

The most plausable explanation seems to be that since there are a large number of interface errors, these are spread equally across all modules and so there are a larger number of errors/1000 executable statements for smaller modules. Some tentative explanations for this behavior are: the majority of the modules examined were small (Table 1) causing a biased result; larger modules were coded with more care than smaller modules because of their size; errors in smaller modules are more apparent and there may indeed still be numerous undetected errors present within the larger modules since all the "paths" within the larger modules may not yet have been fully exercised.

3.7 MODULE COMPLEXITY

Cyclomatic complexity [5] (number of decisions + 1) was correlated with module size. This was done in order to determine whether or not larger modules were less dense or complex than smaller modules containing errors. Scatter plots for executable statments per module versus the cyclomatic complexity were plotted and again, since it was difficult to see any trend in the plots, modules were grouped according to size. The complexity points were obtained by calculating an average complexity measure for each module size class. For example, all the modules which had 50 executable lines of code or less had an average complexity of 6.0. Table 5 gives the average cyclomatic complexity for all modules within each of the size categories. The complexity relationships for executable lines of code within a module is shown in Figure 9. As can be seen from the table the larger modules were more complex than smaller modules.

Module size	Average Cyclomatic Complexity
50	6.0
100	17.9
150	28.1
200	52.7
>200	60.0

AVERAGE CYCLOMATIC COMPLEXITY FOR ALL MODULES
Table 5

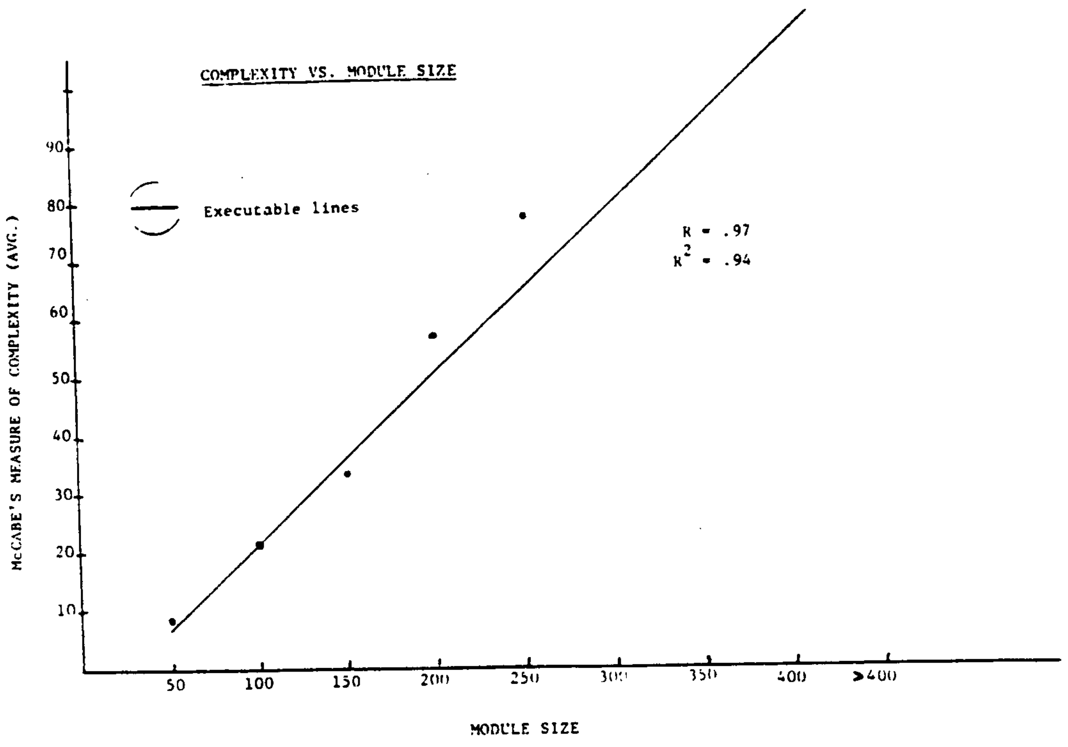


Figure 9

For only those modules containing errors, Table 6 gives the number of errors/1000 executable statements and the average cyclomatic complexity. When this data is compared with Table 5 , one can see that the average complexity of the error prone modules was no greater than the average complexity of the full set of modules.

Module Size	Average Cyclomatic Complexity	Errors/1000 executable lines
50	6.2	65.0
100	19.6	33.3
150	27.5	24.6
200	56.7	13.4
>200	77.5	9.7

COMPLEXITY AND ERROR RATE FOR ERRORED MODULES
Table 6

4.0 DATA NOT EXPLICITLY INCLUDED IN ANALYSES

The 147 modules not included in this study (i.e., assembly segments, common segments, utility routines) contained a total of six errors. These six errors were detected within three different segments. One error occurred in a modified assembly module and was due to the misunderstanding or incorrect statement of the functional specifications for the module. The effort needed to correct this error was minimal (1 hour or less).

The other five errors occurred in two separate new data segments with the major cause of the errors also being related to their specifications. The effort needed to correct these errors was on the average from 1 hour to 1 day (1 day representing 8 hours).

5.0 CONCLUSIONS

The data contained in this paper helps explain and characterize the environment in which the software was developed. It is clear from the data that this was a new application domain in an application with changing requirements.

Modified and new modules were shown to behave similarly except in the types of errors prevalent in each and the amount of effort required to correct an error. Both had a high percentage of interface errors, however, new modules had an equal number of errors of omission and commission and a higher percentage of control errors. Modified modules had a high percentage of errors of commission and a small percentage of errors of omission with a higher percentage of data and initialization errors. Another difference was that modified modules appeared to be more susceptible to errors due to the misunderstanding of the specifications. Misunderstanding of a module's specifications or requirements constituted the majority of errors detected. This duplicates an earlier result of Endres which implies that more work needs to be done on the form and content of the specifications and requirements in order to enable them to be used across applications more effectively.

There were shown to be some disadvantages to modifying an existing module for use instead of creating a new module. Modifying an existing module to meet a similar but different set of specifications reduces the developmental costs of that module. However, the disadvantage to this is that there exists hidden costs. Errors contained in modified modules were found to require more effort to correct than those in new modules, although the two classes contained approximately the same number of errors. The majority of these errors was due to incorrect or misinterpreted specifications for a module. Therefore, there is a tradeoff between minimizing development time and time spent to align a module to new specifications. However, if better specifications could be developed it might reduce the more expensive errors contained within modified modules. In this case, the reuse of "old" modules could be more beneficial in terms of cost and effort since the hidden costs would have been reduced.

One surprising result was that module size did not account for error proneness. In fact, it was quite the contrary, the larger the module the less error prone it was. This was true even though the larger modules were more complex. Additionally, the error prone modules were no more complex across size grouping than the error free modules.

In general, investigations of the type presented in this paper relating error and other change data to the software in which they have occurred is important and relevant. It is the only method by which our knowledge of these types of relationships will ever increase and evolve.

Acknowledgments

The authors would like to thank F. McGarry, NASA Goddard, for his cooperation in supplying the information needed for this study and his helpful suggestions on earlier drafts of this paper.

References

(1) Mendenhall,W. and Ramey,M., Statistics for Psychology, Duxbury Press, North Scituate, Mass., 1973, pp. 280-315.

(2) Endres,A.,"An Analysis of Errors and their Causes in System Programs", Proceedings of the International Conference on Software Engineering, April, 1975, pp. 327-336.

(3) Belady,L.A. and Lehman,M.M., "A Model of Large Program Development", IBM Systems Journal, Vol.15, 1976, pp.225-251.

(4) Weiss,D.M., "Evaluating Software Development by Error Analysis : The Data from the Architecture Research Facility", The Journal of Systems and Software, Vol.1, 1979, pp. 57-70.

(5) Schneidewind,N.F., "An Experiment in Software Error Data Collection and Analysis", IEEE Transactions on Software Engineering , Vol. SE-5, No.3, May 1979, pp.276-286.

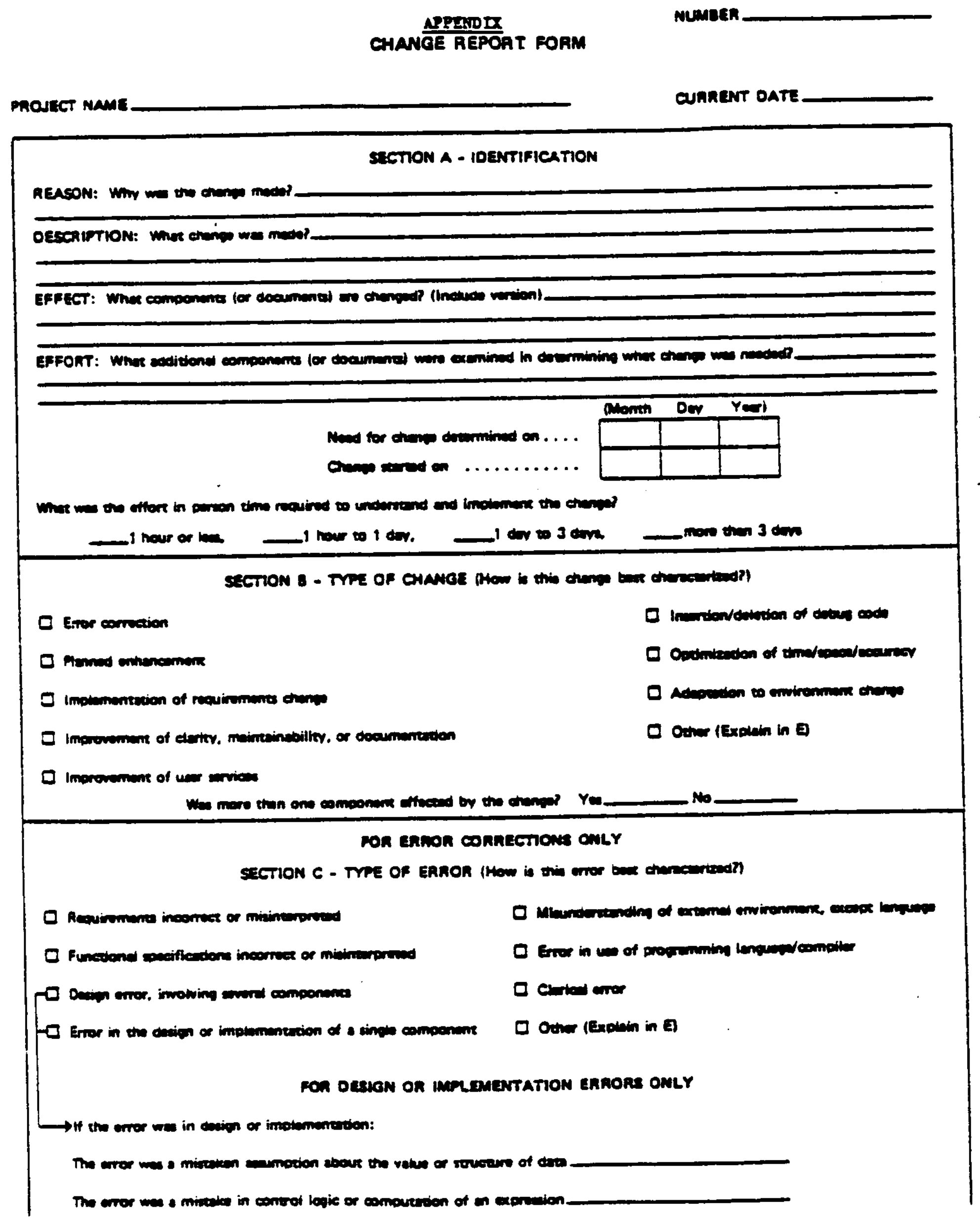
(6) McCabe, T.J., "A Complexity Measure", IEEE Transactions on Software Engineering, Vol. SE-2, No. 4, Dec. 1976, pp.308-320.

(7) Basili,V. and Freburger,K., "Programming Measurement and Estimation in the Software Engineering Laboratory", The Journal of Systems and Software, Vol.2, 1981, pp.47-57.

(8) Weiss, D.M.," Evaluating Software Development by Analysis of Change Data", University of Maryland Technical Report TR-1120, November 1981.

APPENDIX

ORIGINAL PAGE IS
OF POOR QUALITY



APPENDIX
CHANGE REPORT FORM

NUMBER ____________

PROJECT NAME ____________ CURRENT DATE ____________

SECTION A - IDENTIFICATION

REASON: Why was the change made? ____________

DESCRIPTION: What change was made? ____________

EFFECT: What components (or documents) are changed? (Include version) ____________

EFFORT: What additional components (or documents) were examined in determining what change was needed? ____________

	(Month	Day	Year)
Need for change determined on			
Change started on			

What was the effort in person time required to understand and implement the change?

_____1 hour or less, _____1 hour to 1 day, _____1 day to 3 days, _____more than 3 days

SECTION B - TYPE OF CHANGE (How is this change best characterized?)

☐ Error correction
☐ Planned enhancement
☐ Implementation of requirements change
☐ Improvement of clarity, maintainability, or documentation
☐ Improvement of user services
☐ Insertion/deletion of debug code
☐ Optimization of time/space/accuracy
☐ Adaptation to environment change
☐ Other (Explain in E)

Was more than one component affected by the change? Yes ________ No ________

FOR ERROR CORRECTIONS ONLY

SECTION C - TYPE OF ERROR (How is this error best characterized?)

☐ Requirements incorrect or misinterpreted
☐ Functional specifications incorrect or misinterpreted
☐ Design error, involving several components
☐ Error in the design or implementation of a single component
☐ Misunderstanding of external environment, except language
☐ Error in use of programming language/compiler
☐ Clerical error
☐ Other (Explain in E)

FOR DESIGN OR IMPLEMENTATION ERRORS ONLY

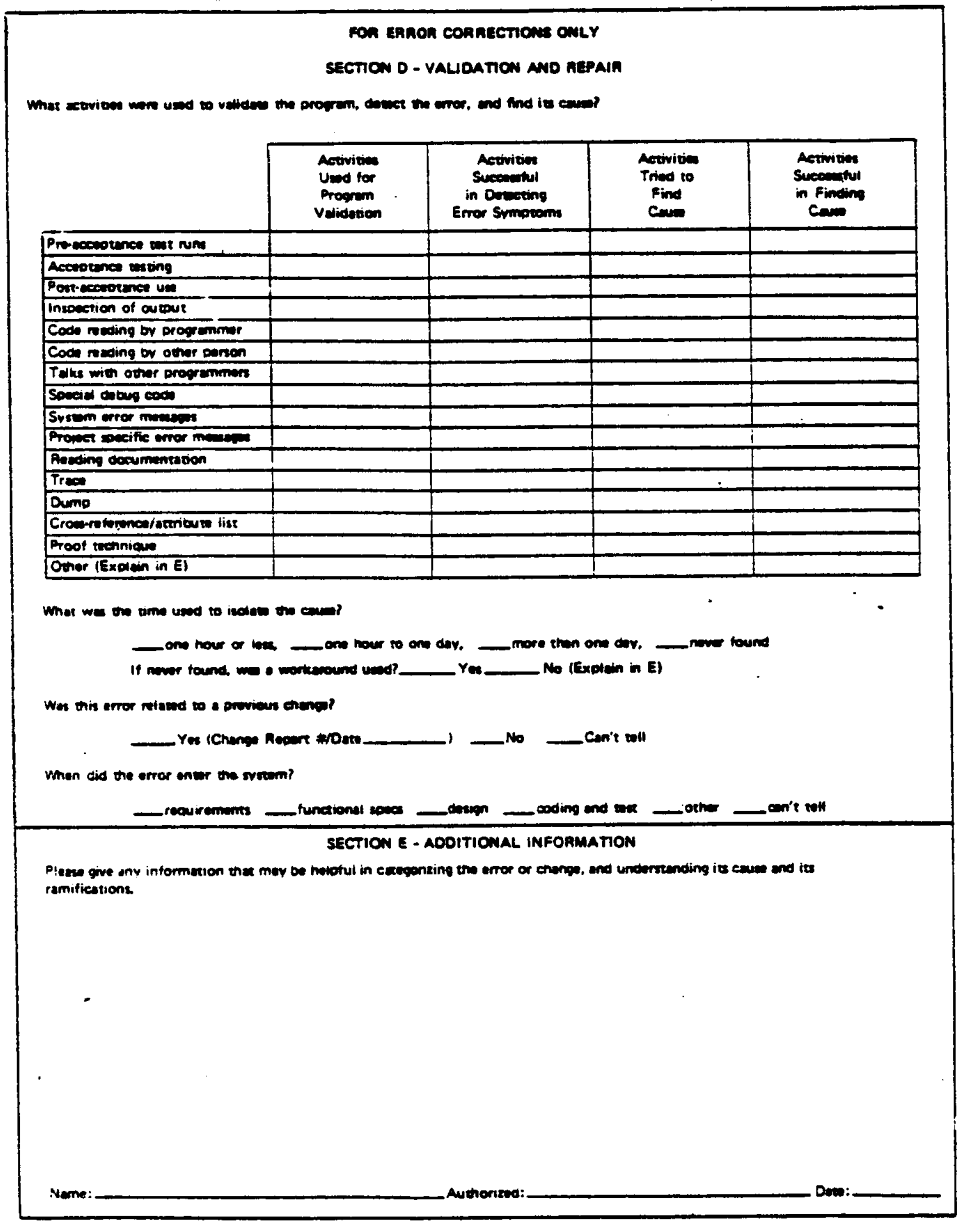
→ If the error was in design or implementation:

The error was a mistaken assumption about the value or structure of data ____________

The error was a mistake in control logic or computation of an expression ____________

Change Report Form

ORIGINAL PAGE IS
OF POOR QUALITY



FOR ERROR CORRECTIONS ONLY

SECTION D - VALIDATION AND REPAIR

What activities were used to validate the program, detect the error, and find its cause?

	Activities Used for Program Validation	Activities Successful in Detecting Error Symptoms	Activities Tried to Find Cause	Activities Successful in Finding Cause
Pre-acceptance test runs				
Acceptance testing				
Post-acceptance use				
Inspection of output				
Code reading by programmer				
Code reading by other person				
Talks with other programmers				
Special debug code				
System error messages				
Project specific error messages				
Reading documentation				
Trace				
Dump				
Cross-reference/attribute list				
Proof technique				
Other (Explain in E)				

What was the time used to isolate the cause?

____one hour or less, ____one hour to one day, ____more than one day, ____never found

If never found, was a workaround used?______Yes______No (Explain in E)

Was this error related to a previous change?

_____Yes (Change Report #/Date__________) ____No ____Can't tell

When did the error enter the system?

____requirements ____functional specs ____design ____coding and test ____other ____can't tell

SECTION E - ADDITIONAL INFORMATION

Please give any information that may be helpful in categorizing the error or change, and understanding its cause and its ramifications.

Name:______________________ Authorized:______________________ Date:________

Change Report Form

SECTION 5 — DATA COLLECTION

SECTION 5 - DATA COLLECTION

The technical papers included in this section were originally prepared as indicated below.

- Basili, V. R., and D. M. Weiss, "A Methodology for Collecting Valid Software Engineering Data," University of Maryland, Technical Report TR-1235, December 1982 (reprinted by permission of the authors)

- Zelkowitz, M. V., "Data Collection and Evaluation for Experimental Computer Science Research," University of Maryland, Technical Memorandum, November 1982 (reprinted by permission of the author)

  A version of this paper also appears in Empirical Foundations for Computer and Information Science (Proceedings), November 1982.

D7-01

N87-24903

Technical Report TR-1235 December 1982
NSG-5123

A METHODOLOGY FOR COLLECTING VALID SOFTWARE ENGINEERING DATA*

Victor R. Basili
University of Maryland

David M. Weiss
Naval Research Laboratory

*Research supported in part by the National Aeronautics and Space Administration Grant NSG-5123. Computer support provided in part by the facilities of NASA/Goddard Space Flight Center and the Computer Science Center at the University of Maryland.

ABSTRACT

An effective data collection method for evaluating software development methodologies and for studying the software development process is described. The method uses goal-directed data collection to evaluate methodologies with respect to the claims made for them. Such claims are used as a basis for defining the goals of the data collection, establishing a list of questions of interest to be answered by data analysis, defining a set of data categorization schemes, and designing a data collection form.

The data to be collected are based on the changes made to the software during development, and are obtained when the changes are made. To insure accuracy of the data, validation is performed concurrently with software development and data collection. Validation is based on interviews with those people supplying the data. Results from using the methodology show that data validation is a necessary part of change data collection. Without it, as much as 50% of the data may be erroneous.

Feasibility of the data collection methodology was demonstrated by applying it to five different projects in two different environments. The application showed that the methodology was both feasible and useful.

A Methodology For Collecting Valid Software Engineering Data

Victor R. Basili

University of Maryland

David M. Weiss

Naval Research Laboratory

I. Introduction

According to the mythology of computer science, the first computer program ever written contained an error. Error detection and error correction are now considered to be the major cost factors in software development [1, 2, 3]. Much current and recent research is devoted to finding ways of preventing software errors. This research includes areas such as requirements definition [4], automatic and semi-automatic program generation [5, 6], functional specification [7], abstract specification [8, 9, 10, 11], procedural specification [12], code specification [13, 14, 15], verification [16, 17, 18], coding techniques [19, 20, 21, 22, 23, 24], error detection [25], testing [26, 27], and language design [16, 28, 29, 30, 31].

One result of this research is that techniques claimed to be effective for preventing errors are in abundance. Unfortunately, there have been few attempts at experimental verification of such claims. The purpose of this paper is to show how to obtain valid data that may be used both to learn more about the software development process and to evaluate software development methodologies in a production environment. Previous [15] and companion papers [32] present the data and evaluation results. The methodology described in this paper was developed as part of studies conducted by the Naval Research Laboratory and by NASA's Software Engineering Laboratory [33].

Software Engineering Experimentation

The course of action in most sciences when faced with a question of opinion is to obtain experimental verification. Software engineering disputes are not usually settled that way. Data from experiments exist, but rarely apply to the question to be settled. There are a number of reasons for this state of affairs. Probably the two most important are the number of potential confounding factors involved in software studies and the expense of attempting to do controlled studies in an industrial environment involving medium or large scale systems.

Rather than attempting controlled studies, we have devised a method for conducting accurate causal analyses in production environments. Causal analyses are efforts to discover the causes of errors and the reasons that changes are made to software. Such analyses are designed to provide some insight into the software development and maintenance processes, help confirm or reject claims made for different methodologies, and lead to better techniques for prevention, detection, and correction of errors. Relatively few examples of this kind of study exist in the literature; some examples are. [34, 35, 4, 15, 36]

To provide useful data, a data collection methodology must display certain attributes. Since much of the data of interest for real projects are collected during the test phase, complete analysis of the data must await project completion. Although it is important that data collection and validation proceed concurrently with development, the final analysis must be done from a historical viewpoint, after the project ends.

Developers can provide data as they make changes during development. In a reasonably well-controlled software development environment, documentation and code are placed under some form of configuration control before being released for use by others than the author. Changes are defined as alterations to baselined design, code or documentation.

A key factor in the data gathering process is validation of the data as they become available. Such validity checks result in corrections to the data that cannot be captured at later times owing to the nature of human memory. [37] Timeliness of both data collection and data validation is quite important to the accuracy of the analysis.

Careful validation means that the data to be collected must be carefully specified, so that those supplying data, those validating data, and those performing the analyses will have a consistent view of the data collected. This is especially important for the purposes of those wishing to repeat studies in both the same and different environments.

Careful specification of the data requires the data collectors to have a clear idea of the goals of the study. Specifying goals is itself an important issue, since, without goals, one runs the risk of collecting unrelated, meaningless data.

To obtain insight into the software development process, the data collectors need to know the kinds of errors committed and the kinds of changes made. To identify troublesome issues, the effort needed to make each change is necessary. For greatest usefulness, one would like to study projects from software production environments involving teams of programmers.

We may summarize the preceding as the following six criteria:

1. the data must contain information permitting identification of the types of errors and changes made,
2. the data must include the cost of making changes and correcting errors,
3. data to be collected must be defined as a result of clear specification of the goals of the study,
4. data should include studies of projects from production environments, involving teams of programmers,
5. data analysis should be historical, but data must be collected and validated concurrently with development
6. data classification schemes to be used must be carefully specified for the sake of repeatability of the study in the same and different environments.

II. Schema For The Investigative Methodology

Our data collection methodology is goal oriented. It starts with a set of goals to be satisfied, uses these to generate a set of questions to be answered, and then proceeds step-by-step through the design and implementation of a data collection and validation mechanism. Analysis of the data yields answers to the questions of interest, and may also yield a new set of questions. The procedure relies heavily on an interactive data validation process; those supplying the data are interviewed for validation purposes concurrently with the software development process. The methodology has been used in two different environments to study five software projects developed by groups with different backgrounds using very different software development methodologies. In both environments it yielded answers to most questions of interest and some insight into the development methodologies used.

The projects studied vary widely with respect to factors such as application, size, development team, methodology, hardware, and support software. Nonetheless, the same basic data collection methodology was applicable everywhere. The schema used has six basic steps, listed in the following, with considerable feedback and iteration occurring at several different places.

1. Establish the goals of the data collection

We divide goals into two categories: those that may be used to evaluate a particular software development methodology relative to the claims made for it, and those that are common to all methodologies to be studied.

As an example, a goal of a particular methodology, such as information hiding [38], might be to develop software that is easy to change. The corresponding data collection goal is to evaluate the success of the developers in meeting this goal, i.e. evaluate the ease with which the software can be changed. Goals in this category may be of more interest to those who are involved in developing or testing a particular methodology, and must be defined cooperatively with them.

A goal that is of interest regardless of the methodology being used is to characterize changes in ways that permit comparisons across projects and environments. Such goals may interest software engineers, programmers, managers, and others more than goals that are specific to the success or failure of a particular methodology.

Consequences of Omitting Goals

Without goals, one is likely to obtain data in which either incomplete patterns or no patterns are discernible. As an example, one goal of an early study [15] was to characterize errors. During data analysis, it became desirable to discover the fraction of errors that were the result of changes made to the software for some reason other than to correct an error. Unfortunately, none of the goals of the study were related to this type of change, and there were no such data available.

2. Develop a list of questions of interest

Once the goals of the study have been established, they may be used to develop a list of questions to be answered by the study. Questions of interest define data parameters and categorizations that permit quantitative analysis of the data. In general, each goal will result in the generation of several different questions of interest. As an example, if the goal is to characterize changes, some corresponding questions of interest are: "What is the distribution of changes according to the reason for the change?", "What is the distribution of changes across system components?", "What is the distribution of effort to design changes?"

As a second example, if the goal is to evaluate the ease with which software can be changed, we may identify questions of interest such as: "Is it clear where a change has to be made in the software?", "Are changes confined to single modules?", "What was the average effort involved in making a change?"

Questions of interest form a bridge between subjectively-determined goals of the study and the quantitative measures to be used in the study. They permit the investigators to determine the quantities that need to be measured and the aspects of the goals that can be measured. As an example, if one is attempting to discover how a design document is being used, one might collect data that show how the document was being used when the need for a change to it was discovered. This may be the only aspect of the document's use that is measurable.

Goals for which questions of interest cannot be formulated and goals that cannot be satisfied because adequate measures cannot be defined may be discarded. Once formulated, questions can be evaluated to determine if they completely cover their associated goals and if they define quantitative measures. Finally, questions of interest have the desirable property of forcing the investigators to consider the data analyses to be performed before any data are collected.

Consequences of Omitting Questions Of Interest

Without questions of interest, there may be no quantitative basis for satisfying the goals of the study. Data distributions that are needed for evaluation purposes, such as the distribution of effort involved in making changes, may have to be constructed in an ad hoc way, and be incomplete or inaccurate.

3. Establish data categories

Once the questions of interest have been established, categorization schemes for the changes and errors to be examined may be constructed. Each question generally induces a categorization scheme. If one question is, "What was the distribution of changes according to the reason for the change?", one will want to classify changes according to the reason they are made. A simple categorization scheme of this sort is *error corrections* vs. *non-error corrections* (hereafter called *modifications*).

Each of these categories may be further subcategorized according to reason. As an example, modifications could be subdivided into those modifications resulting from requirements changes, those resulting from a change in the development support environment (e.g. compiler change), planned enhancements, optimizations, and others.

Such a categorization permits characterization of the changes with respect to the stability of the development environment, with respect to different kinds of development activities, etc. When matched with another categorization such as the difficulty of making changes, this scheme also reveals which changes are the most difficult to make.

Each categorization scheme should be complete and consistent, i.e. every change should fit exactly one of the subcategories of the scheme. To insure completeness, the category "Other" is usually added as a subcategory. Where some changes are not suited to the scheme, the subcategory "Not Applicable" may be used. As an example, if the scheme includes subcategories for different levels of effort in isolating error causes, then errors for which the cause need not be isolated (e.g. clerical errors noticed when reading code) belong in the "Not Applicable" subcategory.

Consequences Of Not Defining Data Categories Before Collecting Data

Omitting the data categorization schemes may result in data that cannot later be identified as fitting any particular categorization. Each change then tends to define its own category, and the result is an overwhelming multiplicity of data categories, with little data in each category.

4. Design and test data collection form

To provide a permanent copy of the data and to reinforce the programmers' memories, a data collection form is used. Form design was one of the trickiest parts of the studies conducted, primarily because forms represent a compromise among conflicting objectives. Typical conflicts are the desire to collect a complete, detailed set of data that may be used to answer a wide range of questions of interest, and the need to minimize the time and effort involved in supplying the data. Satisfying the former leads to large, detailed forms that require much time to fill out. The latter requires a short form organized so that the person supplying the data need only check off boxes.

Including the data suppliers in the form design process is quite beneficial. Complaints by those who must use the form are resolved early (i.e. before data collection begins), the form may be tailored to the needs of the data suppliers (e.g. for use as in configuration management), and the data suppliers feel they are a useful part of the data collection process.

The forms must be constructed so that the data they contain can be used to answer the questions of interest. Several design iterations and test periods are generally needed before a satisfactory design is found.

Our principal goals in form design were to produce a form that:

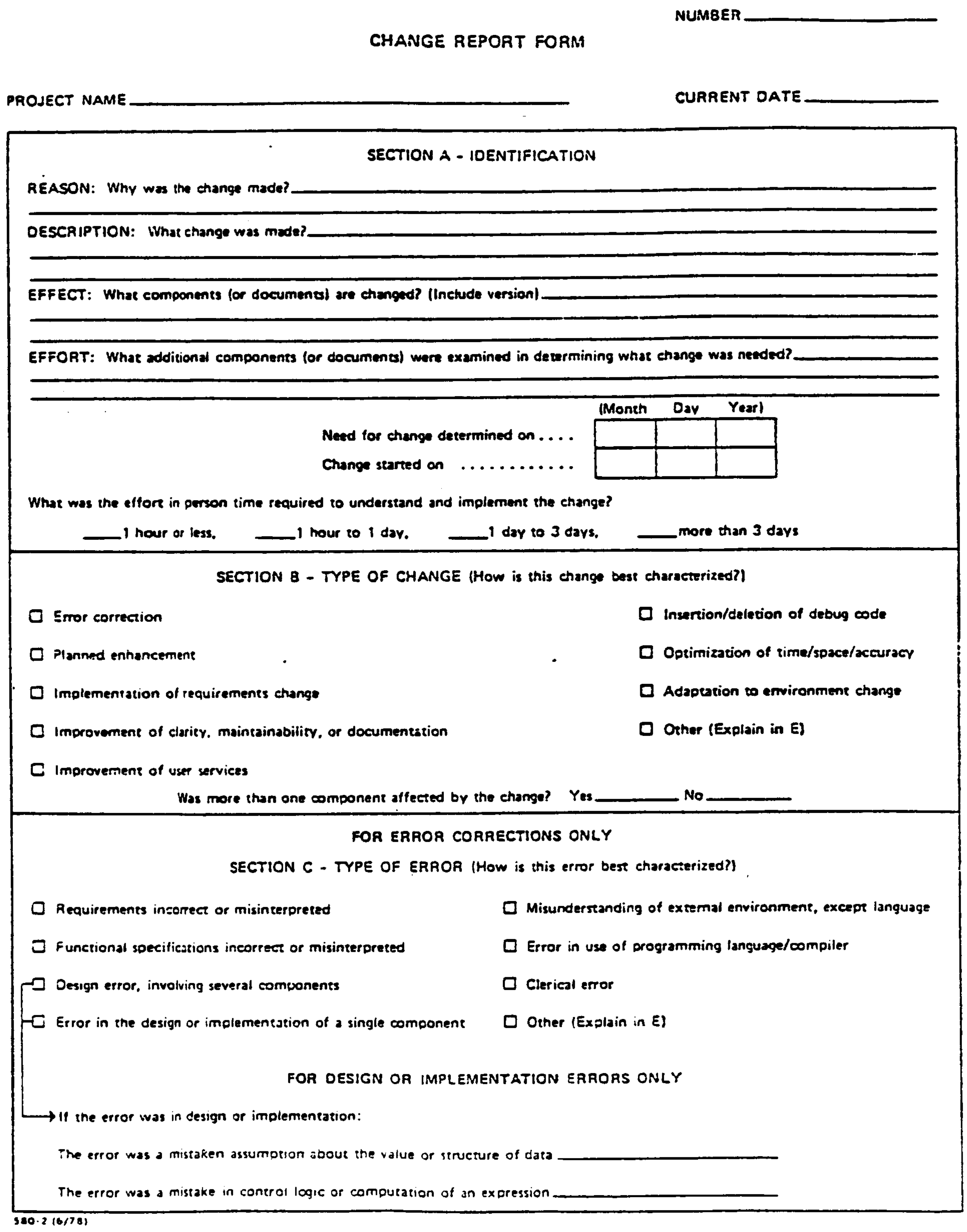
1. fit on one piece of paper,
2. could be used in several different programming environments, and
3. permitted the programmer some flexibility in describing the change.

Figure 1 shows the last version of the form used for the SEL studies. (An earlier version of the form was significantly modified as a result of experience gained in the data collection and analysis processes.) The first sections of the form request textual descriptions of the change and the reason it was made. Following sections contain questions and check-off tables that reflect various categorization schemes.

As an example, a categorization of time to design changes is requested in the first question following the description of the change. The completer of the form is given the choice of 4 categories (one hour or less, one hour to one day, one day to three days, and more than three days) that cover all possibilities for design time.

Consequences Of Not Using A Data Collection Form

Without a data collection form, it is necessary to rely on the developer's memories and on perusal of early versions of design documentation and code to identify and categorize the changes made. This approach leads to incomplete, inaccurate data.



NUMBER ____________

CHANGE REPORT FORM

PROJECT NAME ____________ CURRENT DATE ____________

SECTION A - IDENTIFICATION

REASON: Why was the change made? ____________

DESCRIPTION: What change was made? ____________

EFFECT: What components (or documents) are changed? (Include version) ____________

EFFORT: What additional components (or documents) were examined in determining what change was needed? ____________

	(Month	Day	Year)
Need for change determined on			
Change started on			

What was the effort in person time required to understand and implement the change?

_____1 hour or less, _____1 hour to 1 day, _____1 day to 3 days, _____more than 3 days

SECTION B - TYPE OF CHANGE (How is this change best characterized?)

☐ Error correction	☐ Insertion/deletion of debug code
☐ Planned enhancement	☐ Optimization of time/space/accuracy
☐ Implementation of requirements change	☐ Adaptation to environment change
☐ Improvement of clarity, maintainability, or documentation	☐ Other (Explain in E)
☐ Improvement of user services	

Was more than one component affected by the change? Yes_______ No_______

FOR ERROR CORRECTIONS ONLY

SECTION C - TYPE OF ERROR (How is this error best characterized?)

☐ Requirements incorrect or misinterpreted	☐ Misunderstanding of external environment, except language
☐ Functional specifications incorrect or misinterpreted	☐ Error in use of programming language/compiler
☐ Design error, involving several components	☐ Clerical error
☐ Error in the design or implementation of a single component	☐ Other (Explain in E)

FOR DESIGN OR IMPLEMENTATION ERRORS ONLY

→ If the error was in design or implementation:

The error was a mistaken assumption about the value or structure of data ____________

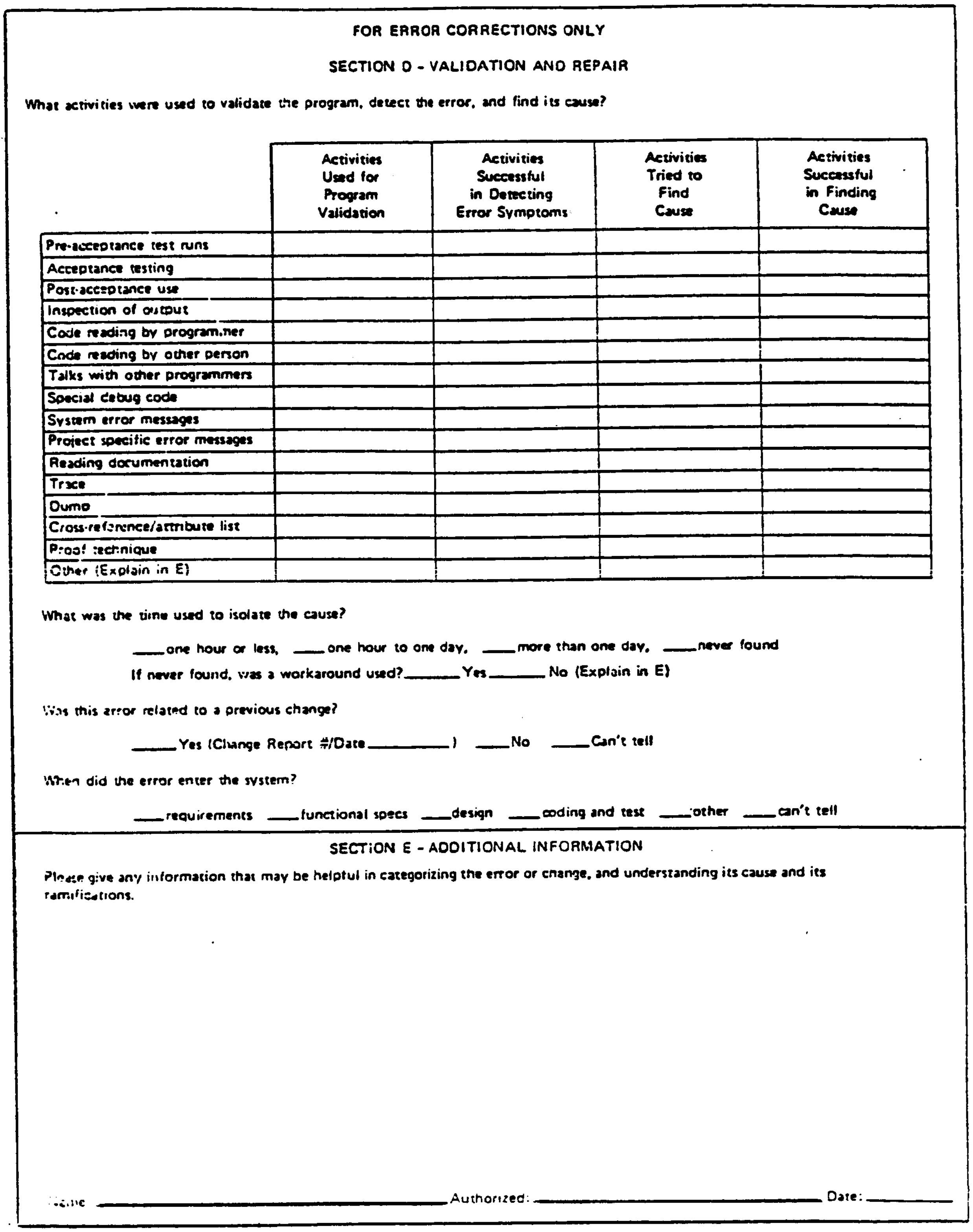
The error was a mistake in control logic or computation of an expression ____________

580-2 (6/78)

Figure 1 SEL Change Report Form (front)

ORIGINAL PAGE IS OF POOR QUALITY

ORIGINAL PAGE IS
OF POOR QUALITY



FOR ERROR CORRECTIONS ONLY

SECTION D - VALIDATION AND REPAIR

What activities were used to validate the program, detect the error, and find its cause?

	Activities Used for Program Validation	Activities Successful in Detecting Error Symptoms	Activities Tried to Find Cause	Activities Successful in Finding Cause
Pre-acceptance test runs				
Acceptance testing				
Post-acceptance use				
Inspection of output				
Code reading by programmer				
Code reading by other person				
Talks with other programmers				
Special debug code				
System error messages				
Project specific error messages				
Reading documentation				
Trace				
Dump				
Cross-reference/attribute list				
Proof technique				
Other (Explain in E)				

What was the time used to isolate the cause?

____one hour or less, ____one hour to one day, ____more than one day, ____never found

If never found, was a workaround used?______Yes______No (Explain in E)

Was this error related to a previous change?

_____Yes (Change Report #/Date_________) ____No ____Can't tell

When did the error enter the system?

____requirements ____functional specs ____design ____coding and test ____other ____can't tell

SECTION E - ADDITIONAL INFORMATION

Please give any information that may be helpful in categorizing the error or change, and understanding its cause and its ramifications.

Name ____________________ Authorized: ____________________ Date: __________

[illegible]

Figure 1 SEL Change Report Form (back)

C-3

5. Collect and validate data

Data are collected by requiring those people who are making software changes to complete a change report form for each change made, as soon as the the change is completed. Validation consists of checking the forms for correctness, consistency, and completeness. As part of the validation process, in cases where such checks reveal problems the people who filled out the forms are interviewed. Both collection and validation are concurrent with software development; the shorter the lag between programmers completing forms and being interviewed concerning those forms, the more accurate the data.

Perhaps the most significant problem during data collection and validation is insuring that the data are complete, i.e. that every change has been described on a form. The better controlled the development process, the easier this is to do. At each stage of the process where configuration control is imposed, change data may be collected. Where projects that we have studied use formal configuration control, we have integrated the configuration control procedures and the data collection procedures, using the same forms for both, and taking advantage of configuration control procedures for validation purposes. Since all changes must be reviewed by a configuration control board in such cases, we are guaranteed capture of all changes, i.e. that our data are complete. Furthermore, the data collection overhead is absorbed into the configuration control overhead, and is not visible as a separate source of irritation to the developers.

Consequences Of Omitting Validation

One result of concurrent development, data collection, and data validation is that the accuracy of the collection process may be quantified. Accuracy may be calculated by observing the number of mistakes made in completing data collection forms. One may then compare, for any data category, pre-validation distributions with post-validation distributions. We call such an analysis a validation analysis. The validation analysis of the SEL data shows that it is possible for inaccuracies on the order of 50% to be introduced by omitting validation. To emphasize the consequences of omitting the validation procedures, we present some of the results of the validation analysis of the SEL data in section III.

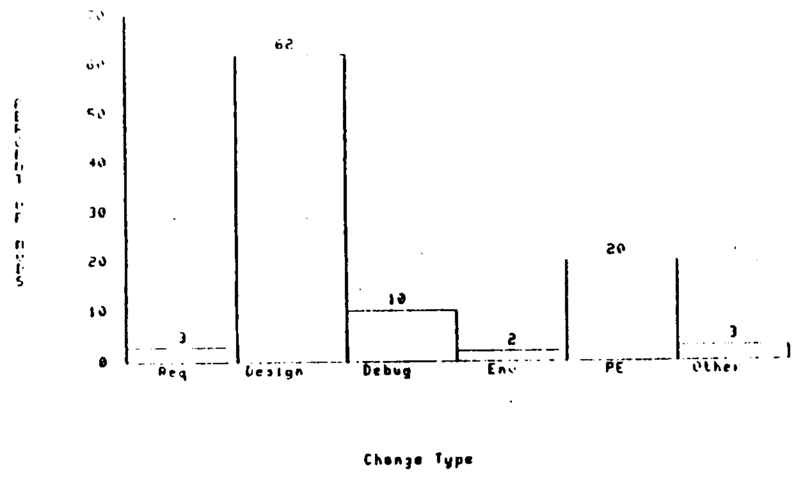
6. Analyze Data

Data are analyzed by calculating the parameters and distributions needed to answer the questions of interest. As an example, to answer the question "What was the distribution of changes according to the reason for the change?", a distribution such as that shown in figure 2 might be computed from the data.

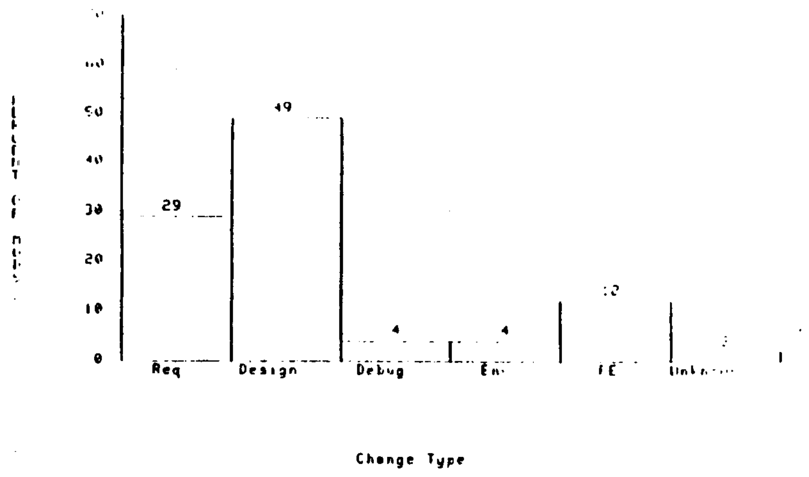
Application of the Schema

Applying the schema requires iterating among the steps several times. Defining the goals and establishing the questions of interest are tightly coupled, as are establishing the questions of interest, designing and testing the form(s), and collecting and validating the data. Many of the considerations involved in implementing and integrating the steps of the schema have been omitted here so that the reader may have an overview of the process. The complete set of goals, questions of interest, and data categorizations for the SEL projects are shown in a companion paper [32].

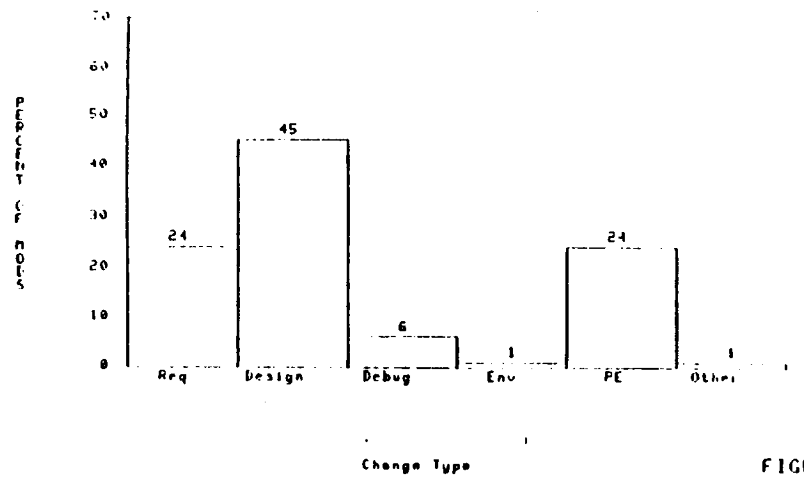
ORIGINAL PAGE IS
OF POOR QUALITY



SEL1



SEL2



SEL3

Key To Figure

Design	Modifications caused by changes in design
Debug	Modifications to insert or delete debug code
Env	Modifications caused by changes in the hardware or software environment
PE	Planned Enhancements
Req	Modifications caused by changes in requirements or functional specifications
Unknown	Causes of these modifications are not known

FIGURE 2 SOURCES OF MODIFICATIONS

Support Procedures and Facilities

In addition to the activities directly involved in the data collection effort, there are a number of support activities and facilities required. Included as support activities are testing the forms, collection, and validation procedures, training the programmers, selecting a data base system to permit easy analysis of the data, encoding and entering data into the data base, and developing analysis programs.

III Details Of SEL Data Collection And Validation

In the SEL environment, program libraries were used to support and control software development. There was a full-time librarian assigned to support SEL projects. All project library changes were routed through the librarian. In general, we define a change to be an alteration to baselined design, code, or documentation. For SEL purposes, only changes to code, and documentation contained in the code, were studied. The program libraries provided a convenient mechanism for identifying changes.

Each time a programmer caused a library change, he was required to complete a change report form (figure 1). The data presented here are drawn from studies of three different SEL projects, denoted SEL1, SEL2, and SEL3. The processing procedures were as follows.

1. Programmers were required to complete change report forms for all changes made to library routines.

2. Programs were kept in the project library during the entire test phase.

3. After a change was made a completed change report form describing the change was submitted. The form was first informally reviewed by the project leader. It was then sent to the SEL library staff to be logged and a unique identifier assigned to it.

4. The change analyst reviewed the form and noted any inconsistencies, omissions, or possible miscategorizations. Any questions the analyst had were resolved in an interview with the programmer. (Occasionally the project leader or system designer was consulted rather than the individual programmer.)

5. The change analyst revised the form as indicated by the results of the programmer interview, and returned it to the library staff for further processing. Revisions often involved cases where several changes were reported on one form. In these cases, the analyst insured that there was only one change reported per form; this often involved filling out new forms. Forms created in this way are known as *generated* forms.

   (Changes were considered to be different if they were made for different reasons, if they were the result of different events, or if they were made at substantially different times (e.g. several weeks apart). As an example, two different requirements amendments would result in two different change reports, even if the changes were made at the same time in the same subroutine.)

Occasionally, one change was reported on several different forms. The forms were then merged into one form, again to insure one and only one change per form. Forms created in this way are known as *combined* forms.

6. The library staff encoded the form for entry into the (automated) SEL data base. A preliminary, automated check of the form was made via a set of data base support programs. This check, mostly syntactic, ensured that the proper kinds of values were encoded into the proper fields, e.g. that an alphabetic character was not entered where an integer was required.

7. The encoded data were entered into the SEL data base.

8. The data were analyzed by a set of programs that computed the necessary distributions to answer the questions of interest.

Many of the reported SEL changes were error corrections. We define an error to be a discrepancy between a specification and its implementation. Although it was not always possible to identify the exact location of an error, it was always possible to identify exactly each error correction. As a result, we generally use the term error to mean error correction.

For data validation purposes, the most important parts of the data collection procedure are the review by the change analyst, and the associated programmer interview to resolve uncertainties about the data.

The SEL validation procedures afforded a good chance to discover whether validation was really necessary; it was possible to count the number of miscategorizations of changes and associated misinformation. These counts were obtained by counting the number of times each question on the form was incorrectly answered.

An example is misclassifications of errors as clerical errors. (Clerical errors were defined as errors that occur in the mechanical translation of an item from one format to another, e.g. from one coding sheet to another, or from one medium to another, e.g. coding sheets to cards.) For one of the SEL projects, 46 errors originally classified as clerical were actually errors of other types. (One of these consisted of the programmer forgetting to include several lines of code in a subroutine. Rather than clerical, this was classified as an error in the design or implementation of a single component of the system.) Initially, this project reported 238 changes, so we may say that about 19% of the original reports were misclassified as clerical errors.

The SEL validation process was not good for verfiying the completeness of the reported data. We cannot tell from the validation studies how many changes were never reported. This weakness can be eliminated by integrating the data collection with stronger configuration control procedures.

Validation Differences Among SEL Projects

As experience was gained in collecting, validating, and analyzing data for the SEL projects, the quality of the data improved significantly, and the validation procedures changed slightly. For SEL1 and SEL2, completed forms were examined and programmers interviewed by a change analyst within a few weeks (typically 3 to 6 weeks) of the time the forms were completed. For project SEL2, the task leader (lead programmer for the project) examined each form before the change analysts saw it.

Project SEL3 was not monitored as closely as SEL1 and SEL2. The task leader, who was the same as for SEL2, by then understood the data categorization schemes quite well and again examined the forms before sending them to the SEL. The forms themselves were redesigned to be simpler but still capture nearly all the same data. Finally, several of the programmers were the same as on project SEL2 and were experienced in completing the forms.

Estimating Inaccuracies In The Data

Although there is no completely objective way to quantify the inaccuracy in the validated data, we believe it to be no more than 5% for SEL1 and SEL 2. By this we mean that no more than 5% of the changes and errors are misclassified in any of the data collection categories. For the major categories, such as whether a change is an error or modification, the type of change, and the type of error, the inaccuracy is probably no more than 3%.

For SEL3, we attempted to quantify the results of the validation procedures more carefully. After validation, forms were categorized according to our confidence in their accuracy. We used four categories:

(1) Those forms for which we had no doubt concerning the accuracy of the data. Forms in this cateogry were estimated to have no more than a 1% chance of inaccuracy.

(2) Those forms for which there was little doubt about the accuracy of the data. Forms in this category were estimated to have at most a 10% chance of an inaccuracy.

(3) Those forms for which there was some uncertaincy about the accuracy, with an estimated inaccuracy rate of no more than 30%.

(4) Those forms for which there was considerable uncertaincy about the accuracy, with an estimated inaccuracy rate of about 50%.

Applying the inaccuracy rates to the number of forms in each category gave us an estimated inaccuracy of at most 3% in the validated forms for SEL3.

Prevalent Mistakes In Completing Forms

Clear patterns of mistakes and misclassifications in completing forms became evident during validation. As an example, programmers on projects SEL1 and SEL2 frequently included more than one change on one form. Often this was a result of the programmers sending the changes to the library as a group.

Comparative Validation Results

Figure 3 provides an overview of the results of the validation process for the 3 SEL projects. The percentage of original forms that had to be corrected as a result of the validation process is shown. As an example, 32% of the originally completed change report forms for SEL3 were corrected as a result of validation. The percentages are based on the number of original forms reported (since some forms were generated, and some combined, the number of changes reported after validation is different than the number reported before validation). Figure 4 shows the fraction of generated forms expressed as a percentage of total validated forms.

Figure 3 shows that pre-validation SEL3 forms were significantly more accurate than the pre-validation SEL1 or SEL2 forms. When the generated and combined forms are also considered, the pre-validation SEL3 data appear to be considerably better then the pre-validation data for either of the other projects. We believe the reasons for this are the improved design of the form, and the familiarity of the task leader and programmers with the data collection process. (Generated forms are shown in figure 4. Combined forms for all of the projects represented a very small fraction of the total validated forms.)

These (overall) results show that careful validation, including programmer interviews, is essential to the accuracy of any study involving change data. Furthermore, it appears that with well-designed forms, and programmer training, there is improvement with time in the accuracy of the data one can obtain. We do not believe that it will ever be possible to dispense entirely with programmer interviews, however.

Erroneous Classifications

Table 1 shows misclassifications of error as modifications and modifications as errors. As an example, for SEL1, 14% of the original forms were classified as modifications, but were actually errors. Without the validation process, considerable inaccuracy would have been introduced into the initial categorization of changes as modifications or errors.

Table 2 is a sampling of other kinds of classification errors that could contribute significantly to inaccuracy in the data. All involve classification of an error into the wrong subcategory. The first row shows errors that were classified by the programmer as clerical, but were later reclassified as a result of the validation process into another category. For SEL1, significant inaccuracy (19%) would be introduced by omitting the validation process.

Table 3 is similar to table 2, but shows misclassifications involving modifications. The first row shows modifications that were classified by the programmer as requirements or specifications changes, but were reclassified as a result of validation.

Variation In Misclassification

Data on misclassifications of change and error type subcategories, such as shown in table 2, tends to vary considerably among both projects and subcategories. (Misclasssification of clerical errors as shown in table 2 is a good example.) This is most likely because the misclassifications represent biases in the judgements of the programmers. It became clear during the validation process that certain programmers tended toward particular misclassifications.

The consistency between projects SEL2 and SEL3 in table 2 probably occurs because both projects had the same task leader, who screened all forms before sending them to the SEL for validation.

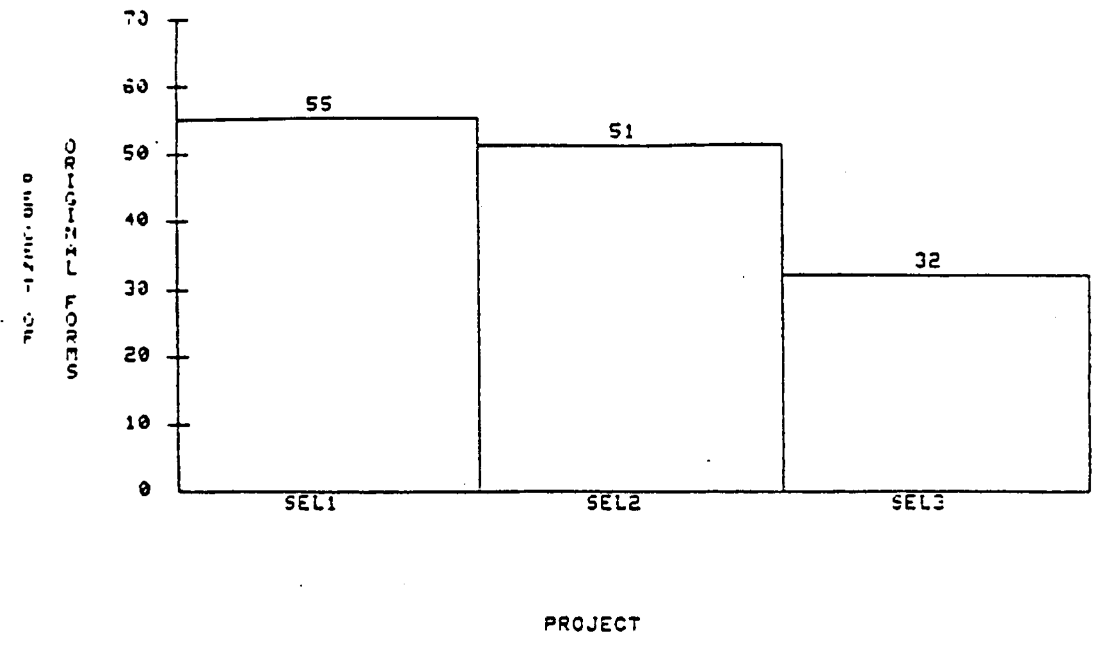


FIGURE 3 CORRECTED FORMS

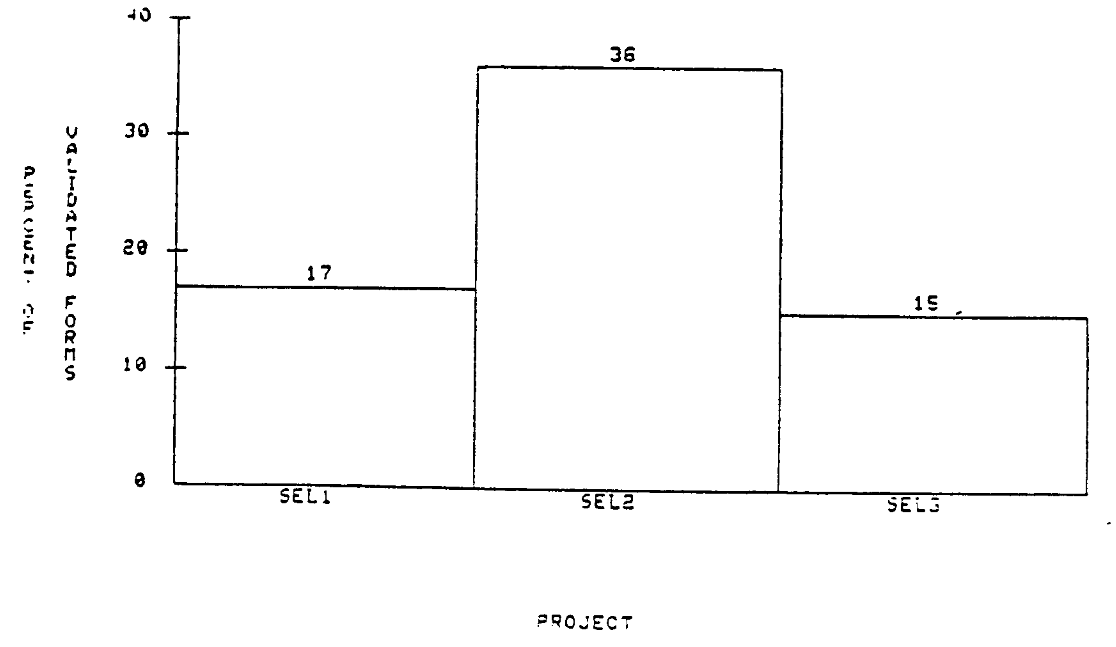


FIGURE 4. GENERATED FORMS

	SEL1	SEL2	SEL3
Modifications classified as errors	1%	5%	less than 1%
Errors classified as modifications	14%	5%	2%

Table 1 Erroneous Modification and Error Classifications
(Percent of Original Forms)

Original Classification	SEL1	SEL2	SEL3
Clerical Error	19%	7%	6%
(Use of) Programming Language	0%	5%	3%
Incorrect or Misinterpreted Requirements		0%	less than 1%
Design Error		8%	1%

Table 2 Typical Error Type Misclassifications
(Percent of Original Forms)

	SEL1	SEL3
Requirements or specification change	1%	less than 1%
Design change	8%	1%
Optimization	8%	less than 1%
Other	3%	less than 1%

Table 3 Erroneous Modification Classifications
(Percent of Original Forms)

Conclusions Concerning Validation

The preceding sections have shown that the validation process, particularly the programmer interviews, are a necessary part of the data collection methodology. Inaccuracies on the order of 50% may be introduced without this form of validation. Furthermore, it appears that with appropriate form design and programmer experience in completing forms, the inaccuracy rate may be substantially reduced, although it is doubtful that it can be reduced to the level where programmer interviews may be omitted from the validation procedures.

A second significant conclusion is that the analysis performed as part of the validation process may be used to guide the data collection project; the analysis results show what data can be reliably and practically collected, and what data cannot be. Data collection goals, questions of interest, and data collection forms may have to be revised accordingly.

IV. Recommendations For Data Collectors

We believe we now have sufficient experience with change data collection to be able to apply it successfully in a wide variety of environments. Although we have been able to make comparisons between the data collected in the two environments we have studied, we would like to make comparisons with a wider variety of environments. Such comparisons will only be possible if more data become available. To encourage the establishment of more data colletion projects, we feel it is important to describe a successful data collection methodology, as we have done in the preceding sections, to point out the pitfalls involved, and to suggest ways of avoiding those pitfalls.

Procedural Lessons Learned

Problems encountered in various procedural aspects of the studies were the most difficult to overcome. Perhaps the most important are the following.

1. Clearly understanding the working environment and specifying the data collection procedures were a key part of conducting the investigation. Misunderstanding by the programmer of the circumstances that require him/her to file a change report form will prejudice the entire effort. Prevention of such misunderstandings can partly be accomplished by training procedures and good forms design, but feedback to the development staff, i.e. those filling out the data collection forms, must not be omitted.
2. Similarly, misunderstanding by the change analyst of the circumstances that required a change to be made will result in misclassifications and erroneous analyses. Our SEL data collection was helped by the use of a change analyst who had previously worked in the NASA environment and understood the application and the development procedures used.
3. Timely data validation through interviews with those responsible for reporting errors and changes was vital, especially during the first few projects to use the forms. Without such validation procedures, data will be severely biased, and the developers will not get the feedback to correct the procedures they are using for reporting data.
4. Minimizing the overhead imposed on the people who were required to complete change reports was an important factor in obtaining complete and accurate data. Increased overhead brought increased reluctance to supply and discuss data. In projects where data collection has been integrated with configuration control, the visible data collection

and validation overhead is significantly decreased, and is no longer an important factor in obtaining complete data. Because configuration control procedures for the SEL environment were informal, we believe we did not capture all SEL changes.

5. In cases where an automated data base is used, data consistency and accuracy checks at or immediately prior to analysis are vital. Errors in encoding data for entry into the data base will otherwise bias the data.

Nonprocedural Lessons Learned

In addition to the procedural problems involved in desinging and implementing a data collection study, we found several other pitfalls that could have strongly affected our results and their interpretation. They are listed in the following.

1. Perhaps the most significant of these pitfalls was the danger of interpreting the results without attempting to understand factors in the environment that might affect the data. As an example, we found a surprisingly small percentage of interface errors on all of the SEL projects. This result was surprising since interfaces are an often-cited source of errors. There was also other evidence in the data that the software was quite amenable to change. In trying to understand these results, we discussed them with the principal designer of the SEL projects (all of which had the same application). It was clear from the discussion that as a result of their experience with the application, the designers had learned what changes to expect to their systems, organized the design so that the expected changes would be easy to make, and then re-used the design from one project to the next. Rather than misinterpreting the data to mean that interfaces were not a significant software problem, we were led to a better understanding of the environment we were studying.
2. A second pitfall was underestimating the resources needed to validate and analyze the data. Understanding the change reports well-enough to conduct meaningful, efficient programmer interviews for validation purposes initially consumed considerable amounts of the change analysts' time. Verifying that the data base was internally consistent, complete, and consistent with the paper copies of reports was a continuing source of frustration and sink for time and effort.
3. A third potential pitfall in data collection is the sensitivity of the data. Programmers and designers sometimes need to be convinced that error data will not be used against them. This did not seem to be a significant problem on the projects studied for a variety of reasons, including management support, processing of the error data by people independent of the project, identifying error reports in the analysis process by number rather than name, informing newly hired project personnel that completion of error reports was considered part of their job, and high project morale. Furthermore, project management did not need error data to evaluate performance.
4. One problem for which there is no simple solution is the Hawthorne (or observer) effect [39]. When project personnel become aware that an aspect of their behavior is being monitored, their behavior will change. If error monitoring is a continuous, long-term activity that is part of the normal scheme of software development, not associated with evaluation of programmer performance, this effect may become insignificant. We believe this was the case with the projects studied.

5. The sensitivity of error data is enhanced in an environment where development is done on contract. Contractors may feel that such data are proprietary. Rules for data collection may have to be contractually specified.

Avoiding Data Collection Pitfalls

In the foregoing sections a number of potential pitfalls in the data collection process have been described. The following list includes suggestions that help avoid some of these pitfalls.

1. Select change analysts who are familiar with the environment, application, project, and development team.
2. Establish the goals of the data collection methodology and define the questions of interest before attempting any data collection. Establishing goals and defining questions should be an iterative process performed in concert with the developers. The developers' interests are then served as well as the data collector's.
3. For initial data collection efforts, keep the set of data collection goals small. Both the volume of data and the time consumed in gathering, validating, and analyzing it will be unexpectedly large.
4. Design the data collection form so that it may be used for configuration control, so that it is tailored to the project(s) being studied, so that the data may be used for comparison purposes, and so that those filling out the forms understand the terminology used. Conduct training sessions in filling out forms for newcomers.
5. Integrate data collection and validation procedures into the configuration control process. Data completeness and accuracy are thereby improved, data collection is unobtrusive, and collection and validation become a part of the normal development procedures. In cases where configuration control is not used or is informal, allocate considerable time to programmer interviews, and, if possible, documentation search and code reading.
6. Automate as much of the data analysis process as possible.

Limitations

It has been previously noted that the main limitation of using a goal-directed data collection approach in a production software environment is the inability to isolate the effects of single factors. For a variety of reasons, controlled experiments that may be used to test hypotheses concerning the effects of single factors do not seem practical. Neither can one expect to use the change data from goal-directed data collection to test such hypotheses.

A second major limitation is that lost data cannot be accurately recaptured. The data collected as a result of these studies represent five years of data collection. During that time there was considerable and continuing consideration given to the appropriate goals and questions of interest. Nonetheless, as data were analyzed, it became clear that there was information that was never requested but that would have been useful. An example is the length of time each error remained in the system. Programmers correcting their own errors, which was the usual case, can supply this data easily at the time they correct the error. Our attempts to discover error entry and removal times after the end of development were fruitless. (Error entry times were particularly difficult to discover.) Given such data, one could isolate errors that were not easily susceptible to detection. This type of example underscores the need for careful planning prior to the start of data collection.

Recommendations That May Be Provided To the Software Developer

The nature of the data collection methodology and the environments in which it can be used do not generally permit isolation of the effects of particular factors on the software development process. The results cannot be used to suggest that controlling a particular factor in the development process will reduce the quantity or cost of particular kinds of errors. We have found that the patterns found in the data do suggest that certain approaches, when applied in the environment studied, will improve the development process.

As an example, in the SEL environment neither external problems, such as requirements changes, nor global problems, such as interface design and specification, were significant. Furthermore, the development environment was quite stable. Most problems were associated with the individual programmer. The data show that in the SEL environment it would clearly pay to impose more control on the process of composing individual routines. Since detecting and correcting most errors was apparently quite easy in the overwhelming majority of cases, more attention should be paid to preventing errors from entering the code initially.

Conclusions Concerning Data Collection For Methodology Evaluation Purposes

The data collection schema presented has been applied to five different projects in two different environments. We have been able to draw the following conclusions as a result of designing and implementing the data collection processes.

1. In all cases, it has been possible to collect data concurrently with the software development process in a software production environment.
2. Data collection may be used to evaluate the application of a particular software development methodology, or simply to learn more about the software development process. In the former case, the better defined the methodology, the more precisely the goals of the data collection may be stated.
3. The better controlled the development process, the more accurate and complete the data.
4. For all projects studied, it has been necessary to validate the data, including interviews with the project developers.
5. As patterns are discerned in the data collected, new questions of interest emerge. These questions may not be answerable with the available data, and may require establishing new goals and questions of interest.

Motivations For Conducting Similar Studies

The difficulties involved in conducting large scale controlled software engineering experiments have as yet prevented evaluations of software development methodologies in the environments where they are often claimed to work best. As a result, software engineers must depend on less formal techniques that can be used in real working environments to establish long-term trends. We view change analysis as one such technique and feel that more techniques, and many more results obtained by applying such techniques, are needed.

Acknowledgments

The authors thank the many people at NASA/GSFC and Computer Sciences Corporation who filled out forms and submitted to interviews, especially Jean Grondalski and Dr. Gerald Page, and the librarians, especially Sam DePriest.

We thank Dr. John Gannon, Dr. Richard Meltzer, Frank McGarry, Dr. Gerald Page, Dr. David Parnas, Dr. John Shore, and Dr. Marvin Zelkowitz for their many helpful suggestions.

Deserving of special mention is Frank McGarry, who had sufficient foresight and confidence to sponsor much of this work and to offer his projects for study.

References

1. B. Boehm and Others, *Information processing/Data Automation Implications Of Air Force Command and Control Requirements in the 1980's (CCIP-85)*, Space and Missile Systems Organization, Los Angeles (February 1972). Technology Trends: Software
2. B. Boehm, "Software and Its Impact: A Quantitative Assessment," *Datamation* **19**(5) pp. 48-59 (May 1973).
3. R. Wolverton, "The Cost Of Developing Large Scale Software," *IEEE Trans. Computers* **23**(6)(1974).
4. T. Bell, D. Bixler, and M. Dyer, "An Extendable Approach to Computer-Aided Software Requirments Engineering," *IEEE Trans. Software Engineering* **SE-3**(1) pp. 49-60 (January 1977).
5. A. Ambler, D. Good, J. Browne, and et. al., "GYPSY: A Language for Specification and Implementation of Verifiable Programs," *Proc. of The ACM Conference on Language Design for Reliable Software*, pp. 1-10 (March 1977).
6. Z. Manna and R. Waldinger, "Synthesis: Dreams => Programs," *IEEE Trans. Software Engineering* **SE-5**(4) pp. 294-329 (July 1979).
7. K. Heninger, "Specifying Requirements for Complex Systems: New Techniques and Their Application," *IEEE Trans. Software Engineering* **SE-6** pp. 2-13 (January 1980).
8. D. L. Parnas, "A Technique For Software Module Specification With Examples," *Comm. ACM* **15**(5) pp. 330-336 (May 1972).
9. J. Guttag, "The Specification and Application to Programming of Abstract Data Types," CSRG-59, University of Toronto Dept. of Computer Science Computer Systems Research Group (1975).
10. J. Guttag, "Abstract data types and the development of data structures," *Comm. ACM* **20** pp. 396-404 (June 1976).
11. B. Liskov and S. Zilles, "Specification Techniques for Data Abstraction," *IEEE Trans. Software Engineering* **SE-1**(1) pp. 7-19 (March 1975).
12. H. Mills, R. Linger, and B. Witt, *Structured Programming Theory and Practice*, Addison-Wesley, Reading (1979).
13. S. Caine and E. Gordon, "PDL - A tool for software design," *Proc. Nat. Computer Conf.*, pp. 271-276 (1975).

14. H. Elovitz, "An Experiment In Software Engineering: The Architecture Research Facility As A Case Study," *Proc. Fourth Intntl Conf. Software Engineering*, pp. 145-152 (1979).

15. D. Weiss, "Evaluating Software Development by Error Analysis: The Data from the Architecture Research Facility," *J. Systems and Software* 1 pp. 57-70 (1979).

16. E. W. Dijkstra, *A Discipline of Programming*, Prentice-Hall, Englewood Cliffs (1976).

17. R. W. Floyd, "Assigning Meanings to Programs," *Proc. Symposium in Applied Mathematics* **XIX** pp. 19-32 American Mathematical Society, (1967).

18. C. A. R. Hoare, "An Axiomatic Basis for Computer Programming," *Comm. ACM* **12**(10) pp. 576-580 (October 1969).

19. F. Baker, "Chief Programmer Team Management of Production Programming," *IBM Systems Journal* **11**(1) pp. 56-73 (1972).

20. E. W. Dijkstra, "Notes on Structured Programming," in *Structured Programming*, Academic Press, London (1972).

21. D. E. Knuth, "Structured Programming With Go To Statements," *Computing Surveys* **6**(4) pp. 261-301 (December 1974).

22. H. Mills, "Chief Programmer Teams: Principles and Procedures," FSC 71-5108, IBM Federal Systems Division (1971).

23. H. Mills, "Mathematical Foundations for Structured Programming," FSC 72-6012, IBM Federal Systems Division (1972).

24. N. Wirth, "Program Development by Stepwise Refinement," *Comm. ACM* **14**(4) pp. 221-227 (April 1971).

25. E. Satterthwaite, "Debugging Tools for High-Level Languages," *Software - Practice and Experience* **2**(3) pp. 197-217 (July-September 1972).

26 W. Howden, "Theoretical and Empirical Studies of Program Testing," *Proc. Third Intntl. Conf. Software Engineering*, pp. 305-310 (May 1978).

27. J. Goodenough and S. Gerhart, "Toward a theory of test data selection," *Proc. Intntl. Conf. Reliable Software*, pp. 493-510 (1975).

28. J. Gannon, "Language Design to Enhance Programming Reliability," CSRG-47, University of Toronto Dept. of Computer Science Computer Systems Research Group (1975).

29. J Gannon and J. Horning, "Language Design for Programming Reliability," *IEEE Trans. Software Eng.* **SE-1**(2)(June 1975).

30. C. A. R. Hoare and N. Wirth, "An Axiomatic Definition of the Programming Language Pascal," *Acta Informatica* **2** pp. 335-355 (1973).

31. K. Jensen and N. Wirth, *Pascal User Manual and Report Second Edition*, Springer-Verlag, New York (1974).

32. V. Basili and D. Weiss, "Evaluating Software Development By Analysis of Changes: The Data From The Software Engineering Laboratory," , ().

33. V. Basili, M. Zelkowitz, F. McGarry, and others, "The Software Engineering Laboratory," Report TR-535, University of Maryland (May 1977).

34 B. Boehm, "An Experiment in Small-Scale Application Software Engineering," Report TRW-SS-80-01, TRW (1980).

35. A. Endres, "Analysis and Causes of Errors in Systems Programs," *Proc. Intntl. Conf. Reliable Software*, pp. 327-336 (1975).

36. V. Basili and D. Weiss, "Evaluation of a Software Requirements Document By Analysis of Change Data," *Proc. Fifth Intntl. Conf. Software Engineering*, pp. 314-323 (March 1981).

37. G. Miller, "The Magical Number Seven, Plus or Minus Two: Some Limits On Our Capacity For Processing Information," *The Psychological Review* **63**(2) pp. 81-97 (March 1956).

38. D. L. Parnas, "On the criteria to be used in decomposing systems into modules," *Comm. ACM* **15**(12) pp. 1053-1058 (December 1972).

39. J. Brown, *The Social Psychology of Industry*, Penguin Books, Baltimore (1954).

D9-61

N87-24904

DATA COLLECTION AND EVALUATION FOR EXPERIMENTAL COMPUTER SCIENCE RESEARCH

Marvin V. Zelkowitz
Department of Computer Science
University of Maryland
College Park, Maryland 20742

Abstract

The Software Engineering Laboratory has been monitoring software development at NASA Goddard Space Flight Center since 1976. This report describes the data collection activities of the Laboratory and some of the difficulties of obtaining reliable data. In addition, the application of this data collection process to a current prototyping experiment is reviewed.

I. INTRODUCTION

There is a significant need to collect reliable data on software development projects in order to provide an empirical basis for making conclusions about software development methodologies, models and tools. However, such data is usually hard to collect and even harder to evaluate. Software is a multibillion dollar industry where 100% cost overruns are common, and maintenance activities can take up to 70% of the total cost of the system [11]. The availability of reliable data to evaluate competing software development techniques is crucial.

As Lord kelvin stated, "I often say that when you can measure what you are speaking about, and express it in numbers, you can know something about it, but when you cannot express it in numbers, your knowledge is of a meager and unsatisfactory kind." The lack of adequate measures is certainly a problem in the software industry today.

Many of the recent analyses of the software development process are based on data that is obtained from university experiments. Students often program special problems whose results are subjected to analysis. This gives the researcher the 10 to 100 data points necessary for statistical validity of the results. However, by virtue of being part of an academic program, such experiments are necessarily small and usually involve inexperienced programmers. There is a need to extend the scope of these experiments to a level appropriate to the multibillion dollar industry.

Most software development data in industry has been collected after the fact. That is, a project is built and then a pile of documents are handed to a research group for evaluation. Often, critical information is missing and the results are not what one would expect. Rather than following the model of archeology - the study of dead software projects, software evaluation must model sociology - the study of living software societies. Data must be collected from ongoing projects, but the software sociologists must not impact the objects of their study. Given the need to finish projects on time and within budgets - a goal too often missed - it is difficult to justify spending money on data collection and evaluation activities.

Specifically to address these problems, the Software Engineering Laboratory (SEL) was set up within NASA Goddard Space Flight Center in 1976. The goal was to study software development activities within NASA and report on experiences that will improve the process. This report describes the SEL and its experiences over the last six years.

II. THE SOFTWARE ENGINEERING LABORATORY

In 1976 the SEL was organized to study software development within the NASA environment. More specifically, its primary charter was to monitor the development of ground support software for unmanned spacecraft. Each such system was typically 30,000 to 50,000 source lines of Fortran and took from 8 to 10 programmers up to two years to build. While this environment is not representative of all software development environments, SEL experiences are generalizable in some respects:

a) Ground support software includes several program types such as data base functions, real time processing, scientific calculations and control language functions. The software is largely implemented in Fortran.

b) By looking at a relatively narrow environment, data collected from many projects can be compared. Thus we get some of the benefits of a carefully controlled experiment without the expense of duplicating large developments. We do not have the problem of looking at a variety of projects, like compilers, COBOL programs, ground support software, MIS programs and then trying to say something consistent about all of these.

To date, 46 projects have been studied, containing over 1.8 million lines of code. Over 150 programmers participated in these projects, and the data base contains over 40 million bytes of data. The general SEL strategy is t carefully monitor a project and regularly collect data during its development. The data is then entered in the SEL data base for analysis. The purpose of this report is not to dwell on specific research results based on this data (See, for example, [8] for a collection of published papers about the SEL) but is concerned with the problems of collecting data, and what we have learned from this process.

III. DATA COLLECTION

III.1 MODEL GENERATION

In order to fully take advantage of the available data, it must be known what information is desired. The models and measures that are to be investigated must be defined. A random data collection activity will usually miss relevant data, and then it will be too late to try and recover that information.

In the SEL, two classes of measures were identified for study, and the data collection activities were oriented around those areas. The initial activities included:

a) Process Measures. Evaluating personnel and computer resources over time was a clear need. One activity was to try and validate models that others have identifie (e.g., the Putnam Norden Rayleigh curve [1]) while another activity was to try and build new models to fit the empirical data (e.g., the Parr curve [7]). Once models were identified, their predictive nature was studied as a means of resource scheduling.

The generation and correction of errors is another activity that has important economic consequences. However, few models are available to build upon, so there was a need to develop new models of errors and investigate their effects upon performance.

b) Product Measures. The size, structure, and complexity of software are other important economic factors to consider. The evaluation of measures such as the software science measures of Halstead [5], the cyclomatic complexity of McCabe [6] and other measures developed within the SEL was another early goal.

Reliability is a critical activity in most environments. In our particular environment, the software that was previously developed was highly reliable (typically under 10 errors in an operational system), so that reliability, while important, was not a primary driving force in organizing the SEL.

III.2 FORMS GENERATION

The first process in evaluating empirical data is the data collection activity. Ideally, you would like the process to be automated and transparent to the programmer. However, this was not possible in this situation. We were interested in the human activities of software development. Thus we needed detailed information about how programmers spend their time. Because of this, a decision made early in the life of the SEL was that some data would be manually collected using a series of forms.

There is a significant tradeoff consideration at this point. If we tried to collect too much information, programmers would object to the interference of the data collection activity on their work. If too little information was asked, then there would be little point in collecting it.

We first developed an initial set of reporting forms. These have been revised several times since then. Each time certain fields were clarified and the amount of information sought decreased somewhat. At the present time, the effort required to fill out the forms is not significant. Initially seven forms were developed. However, only three are used heavily. These seven forms are:

a) Resource Summary. This form lists the number of hours per week spent by all personnel on the project. This information is obtained mostly from the weekly time cards supplied by the contractor. It is easy to obtain this data, and causes little overhead to a project. However, it is very useful for monitoring global resource expenditures, especially in conjunction with the following Component Status Report.

b) Component Status Report. This form is submitted weekly by each programmer. It lists for each component of the system (e.g., Fortran subroutine) the number of hours spent on each of nine categories (e.g., design, code, test, review, etc.). The detail required by this form initially caused some concern; however, in looking over past forms the average programmer worked on only 5 to 10 components per week and only 2 or 3 activities per component. Thus the overhead was not excessive. While the data is only approximate to the nearest hour, we believe that it is more accurate than many other data collection procedures.

For example, many research papers give percentages for design, code, and test on a project. However, these are usually taken from resource summary data and calendar date milestones. If a design review occurs on a Friday, then all activities up until that date are design, with all activities the next week being code. In the SEL environment, there was approximately a 25 percent error in using calendar dates for percent effort [4]. On four projects, approximately 25 percent of the design occurred during the coding phase, while almost half of the testing occurred prior to the testing phase (Figure 1).The Component Status Report is critical for a proper view of development activities.

c) Change Report Form. This form is completed after each change to a component is completed and tested. Due to the number of changes that a component undergoes during early development, there was no attempt to capture this data before the component was "complete" (i.e., through unit test). Note that we are capturing "changes" and not simply "errors." All modifications, due to errors or other considerations such as enhancements, are tracked.

Besides identifying the type of change, this form also identifies the cause of the change - they are not always the same, although programmers have difficulty separating the two. The form also asks for information on the time to find and correct an error, and what tools and techniques were used in the process.

In some environments, the introduction of this form might cause programmers to object; however, this was not the case in our environment. A standard change monitoring procedure was in place, so we simply changed the form that this branch ofNASA GSFC was using before the SEL was created.

These three forms provide the most important data collected by the SEL. Four other forms have been created and used with limited success. These are:

d) Component Summary. This form identifies the characteristics of each component in a system. It gives the size, complexity and interfaces. The goal was to have this form filled out at least twice - once when the component was first identified during design, and again when it was completed. Our experience was that the initial form was filled out before much relevant information was known, and the data on the final form could be extracted automatically from the source code data base.

e) Computer Run Analysis. An entry on this form is filled out for each computer run giving characteristics of the run (execution time, purpose of run, components processed) as well as whether the run met its objectives. This is one form that could be automated. However, the usual range of operating system "Completion Codes" is inadequate for many purposes. For example, a debugging run that was expected to fail at a certain statement, but ran to a successful exit, would have a satisfactory completion code, yet it was a failure as a run since the desired error did not occur.

An interactiv job submittal system could help. Before any run, the system could prompt for some of this information. After the run, the system could ask what happened. Since the current NASA environment consists primarily of interactive editing with batch processing, such an online process would have been difficult to implement.

f) Programmer Analyst Survey. This form attempts to characterize the experiences of the programmers on the project in order to get a general profile of the project tea However, we immediately ran into confidentiality problems concerning personnel records. We never got the detailed information that we desired, but have obtained general comments on each programmer - although the goal is NOT to rate programmers. If there is any hint of any of this data being used for any sort of personnel action, then compliance drops sharply and the value of the data becomes open to question.

g) General Project Summary. This is a form that provides a high-level description of a project. Since the software is developed by NASA and contractor personnel, the form is somewhat superfluous and the information is entered directly into the data base.

An important consideration in forms development is consistency in collecting data. Along with each form a detailed instruction sheet was developed, as well as a glossary of relevant terms like "component," "line of code," and "life cycle phase." For example, we chose the name "component" rather than "subroutine" or "module" simply because those terms were well known (with alternative meanings) and we did not want to evoke any preconceived but wrong image in the minds of the participants. Even so, there was a great deal of confusion about the meanings of the various terms. During the early days of the SEL, many meetings were held to explain the process to programmers. since each programmer worked about one year on a project, after six years there is a large core of personnel experienced in filling out our reporting forms..

III.3 DATA PROCESSING

After being filled out, each form is entered into a data base on a PDP 11/70 computer. In addition to the forms previously described, analyzers were run over the source programs to extract additional information, including lines of code and other measures such as the Halstead software science measures.

Another step in forms processing is data validation. Someone must review the forms as they are submitted. This is expensive, but necessary. It is a quick was to catch and correct errors. In addition, the data entry program should check for data consistency and value ranges. For example, if the program is to read in input in the format MMDDYY, then a month input that is not a number in the range from 01 to 12 must be rejected. A field requiring an input of A, B, or C should reject any other value. Even though we manually check each form, a validation program was more effective for catching errors.

All forms, especially the change report form, need to be reviewed by SEL personnel. Two common errors in the Change report form are to turn in one change report form which actually represented several errors, and the submission of multiple forms for the same error. From earlier work over half of the change report forms were modified following a careful study of each form. This is an expensive process, but needs to be done in order to have accurate data about your environment.

Redundancy of data is another important consideration. Collecting the same or similar data on multiple forms allows for cross validation. There should be a reasonable correlation between the collected values. The resource summary and component status reports have been the easiest to validate. The Computer Run Analysis form is important for validating some of the change report data; however, limited availability of this form has handicapped some of this validation work. Because of that, it is important to manually check each change report form for selected projects.

IV RESEARCH ACTIVITIES

IV.1 PREVIOUS RESEARCH

Research in the SEL has centered on resource and error models and on predicting software productivity. ([8] is a collection of relevant papers published over the last few years.) Perhaps the most important conclusion - although obvious in hindsight - which is relevant to this current discussion is that there is no typical software development environment.

All models include parameters - factors which represent variables in that environment (Figure 2 represents a list of factors from the SEL as well as two other studies [10] [3].) When models based on other environments are applied to the NASA environment, they invariably fail. Does that mean that NASA is different? unique? much better or much worse than other environments? For example, SEL programmers show much higher productivity in lines of code per week than in other organizations. Does that mean that other organizations should pirate away NASA's staff?

Perhaps, but another explanation becomes apparent when NASA's environment is studied in detail. In the SEL, most of the projects are similar ground support software systems. Thus the top level design for these projects are similar. Programmers are experts at this particular problem - thus high productivity. Many factors affecting requirements and design do not apply here. On the other hand, a contractor that bids on a variety of projects - an operating system, a compiler, a data base management system, an attitude orbit determination program, etc. does not build an institutional knowledge about any one particular environment. Requirements and design factors now become significant in this environment and productivity drops.

All companies operate in a different manner. Company policy as to working conditions, computer usage (batch or interactive), leave policy and salaries, management, support tools, etc. all affect productivity. Thus each organization (probably even separate divisions within a single organization) has a different structure and a different set of parameters.

For this reason, one must first calibrate any model to be applied. First develop a quantitative relationship using many factors. Chose those factors relevant to your environment. Calibrate the equations based upon previous projects, and then use the calibrated model for prediction [2]. It is this important calibration step that is missing from most models.

For example, if a baseline equation is given by:

$$\text{Effort} = a * \text{size} + b$$

then one can fit a and b from historical data; and the units of size can be determined from those relevant to your environment - such as lines of code, lines of source (including comments), number of modules, number of output statements, etc. Thus instead of a single model, there is a class of models tailored to each environment.

IV.2 PROTOTYPES

Over the past few years various methodologies have been studied by the SEL. A current SEL activity is the development of software prototypes. Currently software is designed, built and delivered. Rarely is the product evaluated in advance. However, the use of engineering prototypes in a preliminary evaluation is starting to be discussed by software engineering professionals [9].

While the term is appearing with increasing frequency, what does it really mean? Is it a quick and dirty throw-a-way implementation or a carefully designed subset of a final implementation? What are the cost and reliability parameters for a prototype compared to a full implementation.

Currently data on the subject is meagre and usually based on small projects [12]. The SEL is now investigating a larger implementation with some techniques as applied to previous SEL projects.

Briefly, the target implementation is an integrated support system for flight dynamics research. Currently, experimenters (NASA scientists), in trying a new spacecraft model (e.g., a new orbit calculation) must understand the structure of the existing system, access the Fortran source modules, modify them, rebuild the operating program, test it, and then run the experiment - a complex and costly process. The new system is expected to "understand" several flight dynamics systems and to provide a higher level command language that guides the experimenter through the process of building a new version of a system, even if the experimenter is not thoroughly familiar with the existing system. This system is basically a command language interpreter with a complex data dictionary describing the underlying flight dynamics subsystems.

This program is quite different from existing software produced by NASA, so the plan is to prototype it first. Two classes of data will be obtained from the prototype:

a) Characteristics of the process. The Computer Science world has little information available about prototyping, thus this data will add to the general knowledge about this process. What does the life cycle of a prototype look like? How much time is spent in design? code? test? Are errors crucial or can they be side-stepped in the prototype somewhat by "eliminating" the offending feature in the requirements?

Similarly, how does prototyping effect the later full implementation? Will design be easier? Will productivity be higher? Will the overall cost of the system plus prototype be less than the cost of just the full system? Will reliability be higher or the interface more "user friendly?"

b) Predictive nature of the prototype. Once a prototype is built, is it successful? How does one measure success? Will the full system be successful based upon an evaluation of the prototype? A set of measures will be built into the prototype to provide some of these answers.

A baseline study will be made of how experiments are conducted - the cost of machine and people resources will be measured. Some of these experiments will be repeated with the prototype to derive a cost. These will be used to predict the cost of using the full system. If acceptable, then that design will be used for the full implementation, if not, then the design will be modified to correct the problem in the full implementation.

In addition, data will be collected on how often features are used in the prototype, and also how often the prototype is being circumvented in order to provide features that currently do not exist but are needed by the users.

Once the final system is built, the predictive model can be validated in order to aid in developing a theory of software prototypes.

V. CONCLUSIONS

The Software Engineering Laboratory has been in existence for six years and has studied almost 40 projects. The empirical data that has been collected supports several conclusions:

(1) Data collection is hard and expensive. It must be dynamically collected during the development of a project and not after completion.

(2) Data must be validated. Error rates on manually filled out forms are high. A lack of standardized nomenclature for the field hurts consistency. Much effort must go in training personnel to understand the data collection methodology.

(3) Each software development environment is unique. Baseline equations must first be calibrated with past projects before a model can be used in the future.

(4) Little is known, but much is being said, about software prototypes. The SEL is currently studying this issue as part of its ongoing activities.

VI. ACKNOWLEDGEMENTS

This paper was supported by NASA grant NSG-5123 to the University of Maryland. The SEL is under the direction of Frank McGarry of NASA GSFC. Dr. Victor Basili directs the University of Maryland activities of the SEL, and Jerry Page is the coordinator for Computer Sciences Corporation. The results described in this report were developed by the author, the above mentioned persons, as well as several graduate students at the University of Maryland and researchers at CSC. The author is indebted to David Card of CSC for his detailed comments on an earlier draft of this paper.

VII. BIBLIOGRAPHY

[1] Basili V. R. and M. V. Zelkowitz, Analyzing medium scale software developments, Third International Conference on Software Engineering, Atlanta GA, May 1978.

[2] Basili V. R., Models and metrics for software management and engineering, ASME Advances in Computer Technology 1, January, 1980.

[3] Boehm B., Software Engineering Economics, Prentice Hall, 1981.

[4] Chen E. and Zelkowitz M. V., Use of cluster analysis to evaluate software engineering methodologies, Fifth International Conference on Software Engineering, San Diego CA, March, 1981.

[5] Halstead M., *Elements of Software Science,* American Elsevier, 1977.

[6] McCabe T., A complexity measure, *IEEE Transactions on Software Engineering* 2, 1976.

[7] Parr F., An alternative to the Rayleigh Curve model for software development, *IEEE Transactions on software engineering* 6, 1980.

[8] Collected Software Engineering Papers: Volume 1, SEL-82-004, Code 582.1, NASA GSFC, July, 1982.

[9] ACM SIGSOFT Software Engineering Symposium: Workshop on Rapid Prototyping, Columbia, MD, April, 1982.

[10] Walston C. and C. Felix, A method of programming measurement and estimation, *IBM Systems Journal* 16, No. 1, 1977.

[11] Zelkowitz M. V., A. C. Shaw and J. D. Gannon, *Principles of Software Engineering and Design,* Prentice Hall, 1979.

[12] Zelkowitz M. V., A case study in rapid prototyping *Software Practice and Experience* 10, 1037-1042, 1980.

BIBLIOGRAPHY OF SEL LITERATURE

The technical papers, memorandums, and documents listed in this bibliography are organized into two groups. The first group is composed of documents issued by the Software Engineering Laboratory (SEL) during its research and development activities. The second group includes materials that were published elsewhere but pertain to SEL activities.

SEL-ORIGINATED DOCUMENTS

SEL-76-001, Proceedings From the First Summer Software Engineering Workshop, August 1976

SEL-77-001, The Software Engineering Laboratory, V. R. Basili, M. V. Zelkowitz, F. E. McGarry, et al., May 1977

SEL-77-002, Proceedings From the Second Summer Software Engineering Workshop, September 1977

SEL-77-003, Structured FORTRAN Preprocessor (SFORT), B. Chu and D. S. Wilson, September 1977

SEL-77-004, GSFC NAVPAK Design Specifications Languages Study, P. A. Scheffer and C. E. Velez, October 1977

SEL-78-001, FORTRAN Static Source Code Analyzer (SAP) Design and Module Descriptions, E. M. O'Neill, S. R. Waligora, and C. E. Goorevich, February 1978

^{1}SEL-78-002, FORTRAN Static Source Code Analyzer (SAP) User's Guide, E. M. O'Neill, S. R. Waligora, and C. E. Goorevich, February 1978

SEL-78-102, FORTRAN Static Source Code Analyzer Program (SAP) User's Guide (Revision 1), W. J. Decker and W. A. Taylor, September 1982

SEL-78-003, Evaluation of Draper NAVPAK Software Design, K. Tasaki and F. E. McGarry, June 1978

SEL-78-004, Structured FORTRAN Preprocessor (SFORT) PDP-11/70 User's Guide, D. S. Wilson and B. Chu, September 1978

SEL-78-005, Proceedings From the Third Summer Software Engineering Workshop, September 1978

SEL-78-006, GSFC Software Engineering Research Requirements Analysis Study, P. A. Scheffer and C. E. Velez, November 1978

SEL-78-007, Applicability of the Rayleigh Curve to the SEL Environment, T. E. Mapp, December 1978

SEL-79-001, SIMPL-D Data Base Reference Manual, M. V. Zelkowitz, July 1979

SEL-79-002, The Software Engineering Laboratory: Relationship Equations, K. Freburger and V. R. Basili, May 1979

SEL-79-003, Common Software Module Repository (CSMR) System Description and User's Guide, C. E. Goorevich, A. L. Green, and S. R. Waligora, August 1979

SEL-79-004, Evaluation of the Caine, Farber, and Gordon Program Design Language (PDL) in the Goddard Space Flight Center (GSFC) Code 580 Software Design Environment, C. E. Goorevich, A. L. Green, and W. J. Decker, September 1979

SEL-79-005, Proceedings From the Fourth Summer Software Engineering Workshop, November 1979

SEL-80-001, Functional Requirements/Specifications for Code 580 Configuration Analysis Tool (CAT), F. K. Banks, A. L. Green, and C. E. Goorevich, February 1980

SEL-80-002, Multi-Level Expression Design Language-Requirement Level (MEDL-R) System Evaluation, W. J. Decker and C. E. Goorevich, May 1980

SEL-80-003, Multimission Modular Spacecraft Ground Support Software System (MMS/GSSS) State-of-the-Art Computer Systems/Compatibility Study, T. Welden, M. McClellan, and P. Liebertz, May 1980

[1]SEL-80-004, System Description and User's Guide for Code 580 Configuration Analysis Tool (CAT), F. K. Banks, W. J. Decker, J. G. Garrahan, et al., October 1980

SEL-80-104, Configuration Analysis Tool (CAT) System Description and User's Guide (Revision 1), W. Decker and W. Taylor, December 1982

SEL-80-005, A Study of the Musa Reliability Model, A. M. Miller, November 1980

SEL-80-006, Proceedings From the Fifth Annual Software Engineering Workshop, November 1980

SEL-80-007, An Appraisal of Selected Cost/Resource Estimation Models for Software Systems, J. F. Cook and F. E. McGarry, December 1980

[1]SEL-81-001, Guide to Data Collection, V. E. Church, D. N. Card, F. E. McGarry, et al., September 1981

SEL-81-101, Guide to Data Collection, V. E. Church, D. N. Card, F. E. McGarry, et al., August 1982

[1]SEL-81-002, Software Engineering Laboratory (SEL) Data Base Organization and User's Guide, D. C. Wyckoff, G. Page, and F. E. McGarry, September 1981

SEL-81-102, Software Engineering Laboratory (SEL) Data Base Organization and User's Guide Revision 1, P. Lo and D. Wyckoff, July 1983

[1]SEL-81-003, Data Base Maintenance System (DBAM) User's Guide and System Description, D. N. Card, D. C. Wyckoff, and G. Page, September 1981

SEL-81-103, Software Engineering Laboratory (SEL) Data Base Maintenance System (DBAM) User's Guide and System Description, P. Lo and D. Card, July 1983

[1]SEL-81-004, The Software Engineering Laboratory, D. N. Card, F. E. McGarry, G. Page, et al., September 1981

SEL-81-104, The Software Engineering Laboratory, D. N. Card, F. E. McGarry, G. Page, et al., February 1982

[1]SEL-81-005, Standard Approach to Software Development, V. E. Church, F. E. McGarry, G. Page, et al., September 1981

[1]SEL-81-105, Recommended Approach to Software Development, S. Eslinger, F. E. McGarry, and G. Page, May 1982

SEL-81-205, Recommended Approach to Software Development, F. E. McGarry, G. Page, S. Eslinger, et al., April 1983

SEL-81-006, Software Engineering Laboratory (SEL) Document Library (DOCLIB) System Description and User's Guide, W. Taylor and W. J. Decker, December 1981

[1]SEL-81-007, Software Engineering Laboratory (SEL) Compendium of Tools, W. J. Decker, E. J. Smith, A. L. Green, et al., February 1981

SEL-81-107, Software Engineering Laboratory (SEL) Compendium of Tools, W. J. Decker, W. A. Taylor, and E. J. Smith, February 1982

SEL-81-008, Cost and Reliability Estimation Models (CAREM) User's Guide, J. F. Cook and E. Edwards, February 1981

SEL-81-009, Software Engineering Laboratory Programmer Workbench Phase 1 Evaluation, W. J. Decker and F. E. McGarry, March 1981

[1]SEL-81-010, Performance and Evaluation of an Independent Software Verification and Integration Process, G. Page and F. E. McGarry, May 1981

SEL-81-110, Evaluation of an Independent Verification and Validation (IV&V) Methodology for Flight Dynamics, G. Page and F. McGarry, December 1983

SEL-81-011, Evaluating Software Development by Analysis of Change Data, D. M. Weiss, November 1981

SEL-81-012, The Rayleigh Curve As a Model for Effort Distribution Over the Life of Medium Scale Software Systems, G. O. Picasso, December 1981

SEL-81-013, Proceedings From the Sixth Annual Software Engineering Workshop, December 1981

SEL-81-014, Automated Collection of Software Engineering Data in the Software Engineering Laboratory (SEL), A. L. Green, W. J. Decker, and F. E. McGarry, September 1981

SEL-82-001, Evaluation of Management Measures of Software Development, G. Page, D. N. Card, and F. E. McGarry, September 1982, vols. 1 and 2

SEL-82-002, FORTRAN Static Source Code Analyzer Program (SAP) System Description, W. A. Taylor and W. J. Decker, August 1982

SEL-82-003, Software Engineering Laboratory (SEL) Data Base Reporting Software User's Guide and System Description, P. Lo, September 1982

SEL-82-004, Collected Software Engineering Papers: Volume 1, July 1982

[1]SEL-82-005, Glossary of Software Engineering Laboratory Terms, M. G. Rohleder, December 1982

SEL-82-105, Glossary of Software Engineering Laboratory Terms, T. A. Babst, F. E. McGarry, and M. G. Rohleder, October 1983

[1]SEL-82-006, Annotated Bibliography of Software Engineering Laboratory (SEL) Literature, D. N. Card, November 1982

SEL-82-106, Annotated Bibliography of Software Engineering Laboratory Literature, D. N. Card, T. A. Babst, and F. E. McGarry, November 1983

SEL-82-007, Proceedings From the Seventh Annual Software Engineering Workshop, December 1982

SEL-82-008, Evaluating Software Development by Analysis of Changes: The Data From the Software Engineering Laboratory, V. R. Basili and D. M. Weiss, December 1982

SEL-83-001, Software Cost Estimation Experiences, F. E. McGarry, G. Page, D. N. Card, et al., November 1983

SEL-83-002, Measures and Metrics for Software Development, D. N. Card, F. E. McGarry, G. Page, et al., November 1983

SEL-83-003, Collected Software Engineering Papers: Volume II, November 1983

SEL-83-004, SEL Data Base Retrieval System (DARES) User's Guide, T. A. Babst and W. J. Decker, November 1983

SEL-83-005, SEL Data Base Retrieval System (DARES) System Description, P. Lo and W. J. Decker, November 1983

SEL-83-006, Monitoring Software Development Through Dynamic Variables, C. W. Doerflinger, November 1983

SEL-83-007, Proceedings From the Eighth Annual Software Engineering Workshop, November 1983

SEL-RELATED LITERATURE

[2]Agresti, W. W. , F. E. McGarry, D. N. Card, et al., "Measuring Software Technology," Program Transformation and Programming Environments. New York: Springer-Verlag, 1984

[3]Bailey, J. W., and V. R. Basili, "A Meta-Model for Software Development Resource Expenditures," Proceedings of the Fifth International Conference on Software Engineering. New York: Computer Societies Press, 1981

Banks, F. K., "Configuration Analysis Tool (CAT) Design," Computer Sciences Corporation, Technical Memorandum, March 1980

[3]Basili, V. R., "Models and Metrics for Software Management and Engineering," ASME Advances in Computer Technology, January 1980, vol. 1

Basili, V. R., "SEL Relationships for Programming Measurement and Estimation," University of Maryland, Technical Memorandum, October 1979

Basili, V. R., Tutorial on Models and Metrics for Software Management and Engineering. New York: Computer Societies Press, 1980 (also designated SEL-80-008)

[3]Basili, V. R., and J. Beane, "Can the Parr Curve Help With Manpower Distribution and Resource Estimation Problems?", Journal of Systems and Software, February 1981, vol. 2, no. 1

[3]Basili, V. R., and K. Freburger, "Programming Measurement and Estimation in the Software Engineering Laboratory," Journal of Systems and Software, February 1981, vol. 2, no. 1

[2]Basili, V. R., and B. T. Perricone, Software Errors and Complexity: An Empirical Investigation, University of Maryland, Technical Report TR-1195, August 1982

[3]Basili, V. R., and T. Phillips, "Evaluating and Comparing Software Metrics in the Software Engineering Laboratory," Proceedings of the ACM SIGMETRICS Symposium/ Workshop: Quality Metrics, March 1981

[2]Basili, V. R., R. W. Selby, and T. Phillips, "Metric Analysis and Data Validation Across FORTRAN Projects," IEEE Transactions on Software Engineering, November 1983

Basili, V. R., and R. Reiter, "Evaluating Automatable Measures for Software Development," Proceedings of the Workshop on Quantitative Software Models for Reliability, Complexity and Cost, October 1979

[2]Basili, V.R., and D. M. Weiss, A Methodology for Collecting Valid Software Engineering Data, University of Maryland, Technical Report TR-1235, December 1982

Basili, V. R., and M. V. Zelkowitz, "Designing a Software Measurement Experiment," Proceedings of the Software Life Cycle Management Workshop, September 1977

[3]Basili, V. R., and M. V. Zelkowitz, "Operation of the Software Engineering Laboratory," Proceedings of the Second Software Life Cycle Management Workshop, August 1978

[3]Basili, V. R., and M. V. Zelkowitz, "Measuring Software Development Characteristics in the Local Environment," Computers and Structures, August 1978, vol. 10

Basili, V. R., and M. V. Zelkowitz, "Analyzing Medium Scale Software Development," Proceedings of the Third International Conference on Software Engineering. New York: Computer Societies Press, 1978

[3]Basili, V. R., and M. V. Zelkowitz, "The Software Engineering Laboratory: Objectives," Proceedings of the Fifteenth Annual Conference on Computer Personnel Research, August 1977

[2]Card, D. N., "Early Estimation of Resource Expenditures and Program Size," Computer Sciences Corporation, Technical Memorandum, June 1982

[2]Card, D. N., "Comparison of Regression Modeling Techniques for Resource Estimation," Computer Sciences Corporation, Technical Memorandum, November 1982

Card, D. N., and V. E. Church, "Analysis Software Requirements for the Data Retrieval System," Computer Sciences Corporation Technical Memorandum, March 1983

Card, D. N., and V. E. Church, "A Plan of Analysis for Software Engineering Laboratory Data," Computer Sciences Corporation Technical Memorandum, March 1983

Card, D. N., and M. G. Rohleder, "Report of Data Expansion Efforts," Computer Sciences Corporation, Technical Memorandum, September 1982

[3]Chen, E., and M. V. Zelkowitz, "Use of Cluster Analysis To Evaluate Software Engineering Methodologies," Proceedings of the Fifth International Conference on Software Engineering. New York: Computer Societies Press, 1981

[2]Doerflinger, C. W., and V. R. Basili, "Monitoring Software Development Through Dynamic Variables," Proceedings of the Seventh International Computer Software and Applications Conference. New York: Computer Societies Press, 1983

Freburger, K., "A Model of the Software Life Cycle" (paper prepared for the University of Maryland, December 1978)

Higher Order Software, Inc., TR-9, A Demonstration of AXES for NAVPAK, M. Hamilton and S. Zeldin, September 1977 (also designated SEL-77-005)

Hislop, G., "Some Tests of Halstead Measures" (paper prepared for the University of Maryland, December 1978)

Lange, S. F., "A Child's Garden of Complexity Measures" (paper prepared for the University of Maryland, December 1978)

McGarry, F. E., G. Page, and R. D. Werking, Software Development History of the Dynamics Explorer (DE) Attitude Ground Support System (AGSS), June 1983

Miller, A. M., "A Survey of Several Reliability Models" (paper prepared for the University of Maryland, December 1978)

National Aeronautics and Space Administration (NASA), NASA Software Research Technology Workshop (proceedings), March 1980

Page, G., "Software Engineering Course Evaluation," Computer Sciences Corporation, Technical Memorandum, December 1977

Parr, F., and D. Weiss, "Concepts Used in the Change Report Form," NASA, Goddard Space Flight Center, Technical Memorandum, May 1978

Reiter, R. W., "The Nature, Organization, Measurement, and Management of Software Complexity" (paper prepared for the University of Maryland, December 1976)

Scheffer, P. A., and C. E. Velez, "GSFC NAVPAK Design Higher Order Languages Study: Addendum," Martin Marietta Corporation, Technical Memorandum, September 1977

Turner, C., and G. Caron, A Comparison of RADC and NASA/SEL Software Development Data, Data and Analysis Center for Software, Special Publication, May 1981

Turner, C., G. Caron, and G. Brement, NASA/SEL Data Compendium, Data and Analysis Center for Software, Special Publication, April 1981

Weiss, D. M., "Error and Change Analysis," Naval Research Laboratory, Technical Memorandum, December 1977

Williamson, I. M., "Resource Model Testing and Information," Naval Research Laboratory, Technical Memorandum, July 1979

[3]Zelkowitz, M. V., "Resource Estimation for Medium Scale Software Projects," _Proceedings of the Twelfth Conference on the Interface of Statistics and Computer Science_. New York: Computer Societies Press, 1979

[2]Zelkowitz, M. V., "Data Collection and Evaluation for Experimental Computer Science Research," _Empirical Foundations for Computer and Information Science_ (proceedings), November 1982

Zelkowitz, M. V., and V. R. Basili, "Operational Aspects of a Software Measurement Facility," _Proceedings of the Software Life Cycle Management Workshop_, September 1977

---

[1]This document superseded by revised document.

[2]This article also appears in SEL-83-003, _Collected Software Engineering Papers: Volume II_, November 1983.