SOFTWARE ENGINEERING LABORATORY SEL-84-004 SEL-84-004

(NASA-TM-88594) PROCEEDINGS OF THE NINTH ANNUAL SOFTWARE ENGINEERING WORKSHOP (NASA) 361 p HC A16/MF A01 CSCL 09B N86-19967 THRU N86-19980 Unclas G3/61 05491

PROCEEDINGS OF THE NINTH ANNUAL SOFTWARE ENGINEERING WORKSHOP

NOVEMBER 1984

National Aeronautics and Space Administration

Goddard Space Flight Center
Greenbelt, Maryland 20771

PROCEEDINGS

OF

NINTH ANNUAL SOFTWARE ENGINEERING WORKSHOP

Organized by:

Software Engineering Laboratory
GSFC

November 28, 1984

GODDARD SPACE FLIGHT CENTER
Greenbelt, Maryland

CONTENTS

Page

AGENDA iv

SUMMARY OF THE SESSIONS:

Ninth Annual Software Engineering Workshop 1

SESSION 1:

An Approach to Developing Specification Measures 14

Evaluating Software Testing Strategies 42

Software Development in ADA 65

SESSION 2:

A Large Scale Experiment in N-Version Programming 86

Design Metrics for Maintenance 100

An Approach to Operating System Testing 136

The Cognitive Connection: Software Maintenance and Documentation 168

SESSION 3:

An Evaluation of Programmer/Analyst Workstations 178

A Model for the Prediction of Latent Errors Using Data Obtained During the Development Process 196

The Independence of Software Metrics Taken at Different Life-Cycle Stages 213

SESSION 4:

An Interactive Program for Software Reliability Modeling 231

Assessing the Proficiency of Software Developers 264

Tailoring a Software Production Environment for a Large Project 313

Attendance A-1

Bibliography of Sel Literature B-1

OMIT TO
P.14

FOREWARD

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.

Single copies of this document can be obtained by writing to

Frank E. McGarry
Code 552
NASA/GSFC
Greenbelt, Maryland 20771

AGENDA

NINTH ANNUAL SOFTWARE ENGINEERING WORKSHOP
NASA/GODDARD SPACE FLIGHT CENTER
BUILDING 3 AUDITORIUM
NOVEMBER 28, 1984

8:00 a.m.	Registration - 'Sign In' Coffee, Donuts	
8:45 a.m.	INTRODUCTORY REMARKS	J. J. Quann, Deputy Director (NASA/GSFC)
9:00 a.m.	Session No. 1	Topic: Current Research in the Software Engineering Laboratory (SEL) Discussant: F. E. McGarry (NASA/GSFC)
	''An Approach to Developing Specification Measures''	W. Agresti (CSC)
	''Evaluating Software Testing Strategies''	R. Selby (Univ. of Maryland)
	''Analysis of Software Development in Ada''	V. Basili (Univ. of Maryland)
10:30 a.m.	BREAK	
11:00 a.m.	Session No. 2	Topic: Software Error Studies Discussant: M. Zelkowitz (Univ. of Maryland)
	''A Large Scale Experiment In N-Version Programming''	J. Knight (Univ. of Virginia)
	''Design Metrics for Maintenance''	H. Rombach (Univ. of Maryland)
	''An Approach to Operating System Testing''	R. Sum (Univ. of Illinois)
12:30 p.m.	LUNCH	

1:30 p.m.	Session No. 3	Topic: Experiments with Software Development
		Discussant: J. Page (CSC)
	"Implementation and Evaluation of Programmer/Analyst Workstations"	K. Koerner (CSC)
	"A Model for the Prediction of Latent Errors Using Data Obtained During the Development Process"	J. Gaffney (IBM) S. Martello (IBM)
	"The Independence of Software Metrics Taken at Different Life-Cycle Stages"	D. Kafura (Virginia Polytechnical Institute)
3:00 p.m.	BREAK	
3:30 p.m.	Session No. 4	Topic: Software Tools
		Discussant: K. Tasaki (GSFC)
	"An Interactive Program for Software Reliability Modeling"	W. Farr (NSWC)
	"Measuring Proficiency of Software Developers"	L. Putnam (QSM)
	"Tailoring A Software Production Environment of a Large Project"	D. Levine (Intermetrics)
5:00 p.m.	ADJOURN	

NINTH ANNUAL SOFTWARE ENGINEERING WORKSHOP

ABOUT THE WORKSHOP

The Ninth Annual Software Engineering Workshop was held on Nov 28, 1984, at Goddard Space Flight Center in Greenbelt, MD. Nearly 300 people, representing 7 universities, 26 agencies of the federal government, and 56 private organizations, attending the meeting.

As in the past 8 years, the major emphasis for this meeting was the reporting and discussion of experiences in the identification, utilization, and evaluation of software methodologies, models, and tools. Twelve speakers, making up four separate sessions, participated in the meeting with each session having a panel format with heavy participation from the audience.

The workshop is organized by the Software Engineering Laboratory (SEL), whose members represent the NASA/GSFC, University of Maryland, and Computer Sciences Corporation (CSC). The meeting has been an annual event for the past 8 years (1976 to 1984), and there are plans to continue those yearly meetings as long as they are productive.

The record of the meeting is generated by members of the SEL and is printed and distributed by the Goddard Space Flight Center. All persons who are registered on the mail list of the SEL receive copies of the proceedings at no charge.

Additional information about the workshop or about the SEL may be obtained by contacting:

Mr. Frank McGarry
Code 552
NASA/GSFC
Greenbelt, MD 20771

301-344-6846

SUMMARY OF THE SESSIONS: NINTH ANNUAL SOFTWARE ENGINEERING WORKSHOP

Prepared for the

NASA/GSFC

NINTH ANNUAL SOFTWARE ENGINEERING WORKSHOP

By

Q. L. Jordan

COMPUTER SCIENCES CORPORATION

and

THE GODDARD SPACE FLIGHT CENTER

SOFTWARE ENGINEERING LABORATORY

The Ninth Annual Software Engineering Workshop was held on November 28, 1984, at the National Aeronautics and Space Administration (NASA)/Goddard Space Flight Center (GSFC) in Greenbelt, Maryland. This annual fair is held for the purpose of reporting and discussing experiences in measurement, utilization, and evaluation of software methodologies, models, and tools. John J. Quann, Deputy Director of NASA/GSFC, indicated in his opening remarks that NASA's involvement in ever larger and more complex systems, like the space station project, provides a motive for the support of software engineering research and the exchange of ideas in forums such as this workshop. The workshop was organized by the Software Engineering Laboratory (SEL), whose members represent NASA/GSFC, the University of Maryland (UM), and Computer Sciences Corporation (CSC). The workshop was conducted in four sessions that addressed the topics of current SEL research, software error studies, experiments with software development, and software tools. Twelve papers, three for each topic, were presented, with the audience actively participating in all discussions through general commentary, questions, and interaction with speakers.

Approximately 300 persons representing 56 private companies, 7 universities, and 26 agencies of the Federal Government attended the workshop.

One of the major themes of the day pertained to the development, assessment, and verification of software measures applicable to the requirements and design phases of the software life cycle. This theme was addressed by Dr. William Agresti, Dr. Dennis Kafura, Dr. Dieter Rombach, and Mr. John Gaffney. Dr. William Agresti of CSC (An Approach to Developing Specification Measures) discussed the application of a Composite Specifications Model (CSM) that describes specifications from several representative

aspects. The basic purpose of his project was to provide, early in the development process, appropriate information to the diverse groups of managers, analysts, developers, and customers. This information should be provided by objective measures derived from the requirements specification. His paper described one attempt to accomplish this by extracting 29 explicit measures such as number of pages, constraints, and input/output (I/O) requirements from existing requirements specification documents for five NASA/GSFC projects. He showed that, while these measures were extractable, they were not useful. He defined a CSM representing specifications from three aspects: functional (data flow), contextual (entity/relationship), and dynamic (state/transition). Fifty-eight explicit and analytic (i.e., derived from the explicit) measures were defined and extracted from a NASA project that was part of the ground system for a recent shuttle-launched satellite and consisted of 11,000 lines of code. This experiment showed that the CSM is feasible and can provide predictive quantitative information early in development. Since this attempt with the CSM represents only one data point, the CSM must also be applied to other projects. In response to questions, Dr. Agresti indicated that the CSM did not represent performance, so that the traditional specification is not completely replaced. He also noted that tracking changes through several versions is a configuration management problem, but it should be easier with the CSM than the traditional representation.

In another experiment related to the software measures, Dr. Dennis Kafura of Virginia Polytechnic Institute (The Independence of Software Metrics Taken at Different Life Cycle Stages) discussed an effort to define a complete and minimal set of metrics--complete in the sense that all forms

of complexity are represented, and minimal in the sense that no redundant (i.e., highly correlated) measures appear. Three projects, an operating system, a data base system, and a ground support system, were chosen to represent different applications and environments. Metrics considered in the study were broadly classed as code, structure, and hybrid metrics. Code metrics, defined in terms of the implemented code, include Halstead's software science measures and McCabe's cyclomatic complexity. Structure metrics, defined in terms of the relationship between major system components, include Henry and Kafura's information flow complexity and McClure's invocation complexity. Hybrid metrics, combining elements of both code and structure metrics, include Woodfield's syntactic interconnection measure (combining control and data relationships between components with Halstead's effort measure) and Yau and Collofello's stability measure. Study results indicate that the code metrics all seem to be highly correlated. The structure metrics appear to be distinct among themselves and different from code metrics. The relationship of the hybrid to the code and structure metrics is less straightforward. During the following discussion, Dr. Kafura noted that information flow metrics were expensive to obtain, since this involved inputting source code to a tool and working backward. The structure and hybrid metrics might be obtained more easily at design time.

In another effort to assess the utility of measures, Dr. Dieter Rombach of UM (Design Metrics for Maintenance) presented a study to determine the impact of system design characteristics on maintenance behavior. Three timesharing and three process control systems were chosen for the study. The software was characterized by the number of modules and the number of explicit and implicit data structures. The

structure of a module was characterized by exterior complexity (control, data, and information control) and interior complexity (control flow, length, and interface intensity). Dr. Rombach extracted these measures from design documents and then seeded each system with 25 faults. Nine programmers simulated a maintenance environment by responding to the seeded faults, environment changes, and requirements changes. Complexity, stability, and modifiability were compared for different module types, and the results showed that the maintenance behavior of a system can be predicted by an examination of design documents; the best prediction can be obtained from a system that has exterior complexity characterized by integrated information flow.

In yet another effort targeted toward the area of measures, Mr. John Gaffney, Jr., of IBM (A Model for the Prediction of Latent Errors Using Data Obtained During the Development Process) discussed a model implemented on an IBM personal computer that estimates latent (postship) errors on the basis of the count of errors found during each stage of the software life cycle. This technique has proved effective in predicting software errors during late phases of development as well as after system delivery. Model input consists mainly of error counts during each stage of the life cycle, which includes an error discovery process. This process includes high-level design inspections, low-level design inspections, code inspections, unit test, integration test, and system test. A discrete form of the Rayleigh curve is used in the model to represent the number of errors removed per thousand lines of source code (KSLOC) as the independent variable expressed as a function of the error discovery process. This model can be used to aid the management and control of the development process by providing estimates of error counts found during successive stages of the development process. For example, if early error

discovery rates are not as high as predicted, some management action such as additional inspections or test hours may be indicated for later stages to produce an acceptable latent error content. The error discovery histories of different software products and different stages in the development of a single product can be compared.

The second major theme for the day pertained to experimentation with and evaluation of software development methodologies. This theme was addressed by Dr. Richard Selby, Dr. Victor Basili, Ms. Kathy Koerner, and Dr. John Knight. Dr. Richard Selby of UM (Evaluating Software Testing Strategies) described an experiment conducted to compare some common software testing techniques: code reading, functional testing, and structural testing. Thirty-two programmers from NASA/GSFC and CSC participated in the experiment to test three programs. The results of this experiment showed that code reading is more effective in uncovering faults (3.3 errors per hour versus 1.8 errors per hour for the other two methods) and less expensive to utilize than is either functional or structural testing. In the ensuing discussion, Dr. Selby said that no previously unknown errors were found, though some problems reported as errors were cleared up by clarifying requirements or driver programs. He also pointed out that it was not yet clear that the results of this experiment can be generalized to larger programs.

In a second experiment dealing with methodology assessment, Dr. Victor Basili of UM (Analysis of Software Development in Ada*) discussed a project to develop and analyze an Ada

---

*Ada is a registered trademark of the U. S. Government (Ada Joint Program Office).

product in terms of effort and errors. The goals of this project were to evaluate the effect of using Ada for the development methodology, to develop a set of metrics for Ada, and to establish a baseline for future projects using Ada. The experiment task was to redesign and implement in Ada a satellite ground support system that was initially developed in FORTRAN. Four programmers with no prior Ada experience were involved in the redesign and implementation after 1 month of training. Errors were classified as language related (syntax; semantics - i.e., the meaning of an Ada feature; and concept - i.e., how an Ada feature should be used), misunderstanding of the problem or environment, and simple clerical or typographical errors. Dr. Basili noted that the majority of errors found in this project were syntax errors, which led him to conclude that a syntax-directed editor is almost a must with Ada. Programmers tended not to think at a high enough level of abstraction but rather at the FORTRAN code level. Ada features were used, but conservatively, and there was little information hiding. He concluded that training in Ada-based methodology is not only extremely important but also requires a much larger effort than he had originally anticipated. Examples from the area of the given application are needed to understand appropriate data abstraction. Because a higher level of abstraction is required to apply Ada to coding, the development methodology must begin at a higher level early in the development process. In the discussion following Dr. Basili's presentation, the point was made that, if used properly, Ada should result in very high productivity. If, however, Ada is used with the traditional FORTRAN methodology or "mind-set," then developers would do better to stick to FORTRAN.

Ms. Kathy Koerner of CSC (An Evaluation of Programmer/ Analyst Workstations) reported the results of an experiment to evaluate programmer/analyst workstations to automate requirements and design activities. Automation of requirements and design activities promises substantial gains in productivity and quality. CSC and GSFC are conducting a three-step evaluation of programmer/analyst workstations that provides requirements analysis and design tools. The steps are: (1) an assessment of available workstation technology, (2) a controlled experiment utilizing selected workstations, and (3) a long-term study of the effects of workstation use on development. Steps 1 and 2 were completed recently. The industry survey identified four microprocessor implementations of workstations that provided most of the required capabilities. The NASTEC CASE 2000 and Index Technology EXCELERATOR were selected for the in-house evaluation. Both a collection of general users and a division evaluation team participated in the evaluation. The general users rated the EXCELERATOR high with respect to ease of learning and use but otherwise rated the two systems about equally. The division evaluation team rated the CASE 2000 high in terms of overall support. Both systems offer improvements in productivity and quality relative to the manual approach. Differences between the general user and division team evaluations reflect different perspectives on workstation support needs. The EXCELERATOR appears to be better suited for small- to medium-scale projects, while some of the special capabilities of the CASE 2000 make it more attractive for large projects. During Step 3 of the evaluation process, both workstations will be applied to different production projects, and their effects on productivity and quality will be measured objectively. In the following discussion, an estimate was offered for one case

of a 40 percent cost reduction for design and specification in the use of these tools for reworking drawings.

In yet another area of development methodology experimentation, Dr. John Knight of the University of Virginia (A Large-Scale Experiment in N-Version Programming) described a method in which several versions of a program are independently prepared from a single requirements specification, to produce fault-tolerant software. The execution results of all versions which are run with identical input are compared, and a decision is made or output is chosen by vote. Use of the technique implicitly assumes that failures among the several versions are independent. This assumption was tested in an experiment using senior undergraduate and graduate students at the universities of Virginia and California-Irvine. The problem chosen was the development in Pascal of a radar data processor that provided missile friend/foe identification. Twenty-seven Pascal versions of the completed software were subjected to one million tests. Ten versions demonstrated no failures, and most were 99 percent reliable. There were a number of multiple failures. The bugs shared among versions were usually obscure and seemed to result from flaws in problem understanding. The computed probability of multiple failure was 0.000126. However, the observed probability was 0.001255. The independence hypothesis was rejected at the 99 percent confidence level. In response to questions, Dr. Knight added that test data was generated by uniformly distributed random number sequences and that, where a parameter had a range, the value was varied throughout the range. No testing of real-time capabilities was done.

The third group of presenters covered a potpourri of topics ranging from productivity to configuration management. The presenters were Mr. Larry Putnam, Mr. David Levine, Mr. R. N. Sum, Mr. William Farr, and Mr. Oliver Smith.

Specific topics were empirical studies to model productivity and other development characteristics as functions of staffing profiles, the impact of formalized and automated configuration tools for large-scale development projects, a heuristic method for testing an operating system, and an interactive tool to support software reliability modeling. Mr. Larry Putnam of Quantitative Software Management, Inc. (QSM) (Measuring Proficiency of Software Developers) presented the results of empirical studies he has performed to model productivity and other software development characteristics in terms of staffing profiles, and he pointed out sharp differences between development in the United States and Japan. An algorithm from the Software Lifecycle Management Model (SLIM) was used to develop a productivity index to measure the proficiency of software developers. This productivity index has been computed for each of the 800 systems in the QSM data base. Comparisons of the developer performance were made with development time and effort, project staffing, and productivity. From this analysis, it was possible to determine the developers' style for building software. One style was characterized by fast buildup of resources, high staffing levels, and quick product delivery. Another style was characterized by slower buildup, lower staffing levels, and slower product delivery. Three Japanese companies exhibiting the former style were contrasted with three American companies exhibiting the latter style. The Japanese cost was higher and productivity was lower than the U.S. companies. The implication is that scheduling and staffing, controlled by management, have a significant impact on productivity. During the following discussion, Mr. Putnam said that the slower buildup development style produced higher quality code. One U.S. manufacturer got five times better code.

Mr. David Levine of Intermetrics, Inc. (Tailoring a Software Environment for a Large Project) described the impact of utilizing automated and formalized configuration control tools in the support of disciplined development for large-scale projects. A software production environment was constructed to meet the goals of a specific large programming project (100 KSLOC and 700 modules). A method was developed to automatically maintain the version identification of each module in a form that was easily visible and checkable by standard tools, especially by the linker. The version number was also appended to a module when copied into the programmer's private library. The version number was then frozen and was carried into the object code and load modules. The development language supported separate compilation. This capability required good management to maintain correctness and to control recompilation. A system was developed in which the interface definitions were provided in the same files as the functions they described. They could then be extracted for inclusion by other units. Other systems were developed to meet the needs imposed on the project by continuous integration to maintain a stable official baseline configuration while developers were adding and modifying code. The environment was implemented on UNIX to support development by up to 20 programmers. The project took 2 years and involved 9200 versions. A project of this size seems to require a less strong interconnection and less changeable interface than a smaller project. This has major implications for the support system.

Mr. R. N. Sum, Jr., of the University of Illinois (An Approach to Operating System Testing) discussed a heuristic method used to test an operating system. The results of applying this method to the IBM System 9000 XENIX operating system test and the development of a UNIX test suite were presented. System specifications were used to divide the

system into manageable pieces to test, and user's manuals were used to develop the specific tests. The system was divided into high-level commands available to users, library and utility subroutines, and system calls used by the system programmers and drivers. Testing methods applied were exhaustive (every possible value in the input range), random (values randomly selected from the input range), special (specific values of input that have specific or unusual results), explicit (values explicitly used or suggested in the manuals), and exception (illegal input values to test error handling). A Problem Tracking Memorandum (PTM) was used to document errors. Commands, being the largest category, were the most error prone (51.9 percent of PTMs), with documentation accounting for 19.6 percent of the PTMs. A surprising 15.4 percent of PTMs were accounted for by system calls. Results were also presented by test type, error severity level, and manpower profile. The method exhibits many of the characteristics of a good system test, and even though the System 9000 is considered a small system, the system test used approximately 30 research-assistant months. It therefore appears that hardware advancements are blurring the concept of size so that size must be carefully considered in system development.

Mr. William Farr and Mr. Oliver Smith of the Naval Surface Weapons Center (An Interactive Program for Software Reliability Modeling) described an interactive tool that has been developed in support of the use of several well-known software reliability models for the estimation and analysis of errors. They implemented a Statistical Modeling and Estimation of Reliability Functions for Software (SMERFS) to facilitate the application of reliability analysis. The program includes eight well-known models, four based on error interarrival time and four based on the count of errors per testing period. Development goals of the SMERFS

program were maintainability, providing a complete reliability environment, interactive capability, error detection capability, and machine transportability. The use of the program was illustrated with a sample data analysis. The user may, by menu selection, input, edit, or transform data, obtain statistics and plots of input data, run a model from the choice of eight, obtain goodness-of-fit results, and generate plots of original and predicted data and plots of residual data. In the discussion that followed the presentation, Mr. Farr said that the program has been used by two large-scale Navy projects with very good results. Predicted error rates using some of these models were within 10 percent of actual.

PANEL #1

THE SOFTWARE ENGINEERING LABORATORY (SEL)

W. Agresti, Computer Sciences Corp
R. Selby, University of Maryland
V. Basili, University of Maryland

D1
N86-19968

AN APPROACH TO DEVELOPING SPECIFICATION MEASURES[1]

William W. Agresti[2]
Computer Sciences Corporation

ABSTRACT

An approach to developing specification measures is described. A key feature of the approach is the introduction of a new requirements representation, the Composite Specification Model (CSM). Results are reported from an experiment in which the requirements for a real system are recast using the CSM. Specification measures are then extracted from the CSM representation of the system.

[1]Proceedings, Ninth Annual Software Engineering Workshop, National Aeronautics and Space Administration, Goddard Space Flight Center, November 1984

[2]Author's Address: System Sciences Division, Computer Sciences Corporation, 8728 Colesville Road, Silver Spring, Maryland 20910

INTRODUCTION

The first objective of the Software Engineering Laboratory (SEL) (Reference 1) is to understand the software development process in the flight dynamics environment of the National Aeronautics and Space Administration (NASA) Goddard Space Flight Center (GSFC). To meet this objective, many SEL studies (e.g., References 2 through 4) have followed Lord Kelvin's admonition (Reference 5) that "satisfactory" understanding comes through measurement. However, aspects of the software development process and product accommodate measurement to vastly different degrees. Coding and testing, for example, lead to familiar measures such as lines of code and fault rate. But if we want measures that will help us estimate and plan, these measures become available too late in the software development life cycle to be of use. In earlier phases, measurement grows increasingly more difficult. As the target of measurement shifts from coding to design and, ultimately, to requirements, we find that the familiar measures depend on information that is no longer available. Despite this expected difficulty, this study sought to extend the SEL's measurement horizon to the requirements phase.

THE MEASUREMENT OF REQUIREMENTS SPECIFICATIONS

One way to account for the difficulty in measuring requirements is to recognize the needs of the various audiences who use requirements specifications.

People who fulfill four different roles--analyst, manager, developer and customer--look to the requirements for different reasons. While it is axiomatic to say that all four groups want a "good" specification, closer inspection reveals the "goodness" taking many forms. Figure 2* implies that a good specification possesses certain desirable properties e.g., consistency, completeness, and understandability.

*All figures are grouped together at the end of the paper.

Figure 2 also suggests that a good specification facilitates assessment and estimation:

- How complex are the requirements?
- How much will it cost to develop software that satisfies the requirements?
- How familiar is this application to our development staff?

The measurement goal is to encase the requirements in a shell (Figure 3) so that anyone referring to the specification may now obtain a measure of his/her property of interest. Clearly the requirements specification will need to be processed in some fashion to generate such property measures.

THE INITIAL APPROACH

The approach (Figure 4) to providing specification measures was driven by a preference for objective measures instead of questionnaires or other subjective ratings (Reference 6). Requirements specifications from the flight dynamics area were examined for the purpose of identifying measurable attributes. A total of 29 measures were defined (Reference 7).

Because of the interest in objectivity, the resulting measures were explicit counts--number of pages, number of constraints, etc.--that were believed to be unaffected by the analyst extracting the measures.

As an experiment, the measures were extracted from several requirements specifications. Being explicit counts, the measures were easy to extract. However, examination of the metric values led to the conclusion that they were not useful for quantitatively characterizing the requirements.

This conclusion is not a judgment on the contents of the requirements documents. Rather, it finds that the requirements specifications do not facilitate objective measurement. Such a result is not unexpected. Boehm has observed (Reference 8) that "Some work has been done to correlate the amount of software development effort to the number of specification elements. . . . These attempts have run into the same sort of definitional and normalization problems as have the 'number of routines, reports, etc.'...."

Figure 5 is an example of the extracted data that led to the conclusion. Five flight dynamics projects were selected

because all involved spacecraft attitude determination. Furthermore, their requirements documents contained identical section headings, indicative of parallel organization. Because of the commonality in the documents' structure, the measures might be more apt to exhibit some pattern that can be exploited to advantage for estimation or property detection.

Figure 5 depicts, for the five projects, the number of new source lines of code in the delivered system plotted against the number of pages in the system's requirements document. The intuition is simple: if it takes more pages to specify the requirements for one system than another, then we would expect the first system to be larger than the second because both systems were built to satisfy their requirements specifications.

The scatter in Figure 5 shows that our expected pattern did not materialize. The requirements documents for projects D and E, for example, were nearly the same size, but project E had five times the number of new source lines as project D. Other evidence that the extracted metric values were not true indicators of the requirements was not easy to display graphically. Measures such as "number of constraints" were difficult to enumerate fairly when aspects of the requirements were expressed at different levels of detail.

The lesson learned from this initial excursion into requirements specification metrics was that "representation is everything!" The simple counts we extracted were not useful measures because they reflect the variability that is found in the representation of requirements.

THE REVISED APPROACH

The message was clear: get the process of representing requirements under control. Only then would we have confidence that our extracted measures were indicative of the underlying requirements and not an artifact of their textual representation.

Our revised approach centered on the development of a different requirements representation, one that would enable the definition and extraction of objective measures. We proposed a five-step plan (Figure 6) that included an experiment of applying the new model to a real system.

The first step the revised procedure was to propose a new representation. We sought a representation that would accommodate the varying sized projects that are found in the flight dynamics environment. Requirements statement languages were an alternative. However, previous SEL experience (Reference 9) with such languages suggested their use only for larger projects, rather than those common to our environment (Reference 1).

We developed a representation called the Composite Specification Model, or CSM. It seemed both realistic and valuable as a template for specifying requirements. CSM is motivated by the work of DeMarco and others (References 10 and 11) in expressing the benefits of multiple views of requirements. The inherent complex behavior of large software and the multiple audiences for requirements specifications (Figure 2) support the observation that no single view of the requirements will be satisfactory. DeMarco (Reference 10) suggests an analogy to this situation is a three-dimensional object presented in a two-dimensional medium: an illustration would show the orthogonal projections of the object onto each plane.

Another analogy is the representation of a building. The architect may use a scale model to show the planning commission and a set of blueprints to show the electricians. More than one representation of a complex object may exist at any time. The object's features that are highlighted depend on the needs of the audience.

STEP 1: THE COMPOSITE SPECIFICATIONS MODEL

The CSM is a composite of different viewpoints, each with its own notation (Figure 7). Currently, the CSM is comprised of three views--functional, contextual, and dynamic--but more could be added. The decision was made to advocate distinct "pure" views as opposed to embellishing an existing notation (e.g., data flow diagrams) with new symbols and associated semantics. Generating the CSM would impose a healthy discipline on the analyst to briefly restrict his or her attention, for example, to functional issues. The analyst would capture that understanding of functional requirements in a notation before moving on to consider, in turn, the contextual and dynamic views.

Certain properties of the CSM are significant. First, the number of views is not fixed at three; more may be added. Second, the viewpoint is not ultimately connected to one specific notation. If a better notation were found for the dynamic view, for example, it could be introduced. In this sense, the CSM can grow and adjust to new developments.

The current notations for the CSM are

- Data flow analysis (for functional view)
- Entity-relationship (ER) approach (for contextual view)
- State-transition analysis (for dynamic view)

All three notations may be expressed using diagrams, making the CSM more accessible because of its nonnarrative style.

Examples of the three views are presented in Figures 8, 9, and 10.

Figure 8 represents a data flow diagram of processes, data flows, and data stores in accordance with the guidelines in Reference 12. A data dictionary would accompany the diagrams to provide definitions of the data items, data records, external entities, processes, data flows, and data stores. Because data flow analysis is generally well known, it will not be discussed further; References 12 may be consulted for a detailed introduction.

While functional processing is a predictable component of most specification models, the contextual view is not so obvious a choice. Hence, the motivation for its use will be discussed. The environment or information space in which the system will reside is of immediate concern. Capturing the context of a system has been relatively undervalued as a tool for requirements engineering. A partial explanation may be that, for small programming exercises (e.g., sorting numbers or solving an equation), the background environment is either nonexistent or not a major concern, and therefore needs no representation. Many of the guidelines for addressing large system development have begun as attempts to "scale-up" the approaches (e.g., structured techniques) that were successful with small programs. Because the context is not important in understanding small programs, it has not been one of the techniques that investigators pursued in this scaling-up process.

With larger systems, the context or environment is a significant element in understanding the system's behavior. The software system is modeling some portion of an environment. The system, when it is completed, will be taking its place in that environment, interacting with other objects (e.g., hardware, sensors, other software) that are producing behavior in the environment. To describe its behavior relative to these other objects, the system must refer to specific attributes of the objects, for example, the mean radius of

the Earth or the size of fuel tanks. Likewise, events in the environment (e.g., loss of signal, thruster on-time) may trigger behavior by the system. Not all of the attributes or events in the environment are modeled by the system. In this sense, the model of the environment is not complete, nor is it ever intended to be complete. An individual attempting to understand the functioning and behavior of the software will be aided by seeing a representation of precisely those objects, attributes, and events that the system needs to know about in its environment.

The representation of the environment is not the same as a data dictionary. Data items in the dictionary may have no counterpart in the breakdown of objects, attributes, and events in the environment. Conversely, descriptors in the environment (e.g., Earth, gyro) will not always correspond to data items.

Because of the increase in complexity, it is much more difficult to specify the requirements for large systems than for small programs. Simon (Reference 13) sees the origin of the added complexity in a rough analogy between large systems and humans as decisionmaking, behavior-producing entities:

> "A man, viewed as a behaving system is quite simple. The apparent complexity of his behavior over time is largely a reflection of the complexity of the environment in which he finds himself."

The implication is that a large system is more complex because it is modeling more of a complex environment. In this sense, representing the environment in the CSM requires focusing properly on the source of the complexity.

Capturing the information space or context will be extremely valuable in making decisions about the reusability of systems. From this representation, the particular environment of an existing system will be visible. An analyst or developer will thus be able to assess the degree of reusability based on the new system's similarity to the objects, attributes, and events characterizing the environment of an existing system.

Modifiability or designing for change is a desirable attribute of a system. Its embodiment earlier in the life cycle is to "specify for change." Many of the changes to a system are responses to changes in the environment. When the specification includes a representation of the environment, the effects of such changes are easier to assess, because both the change and the specification being changed are expressed

in the same terms in the domain of the application and the user.

The form used in the CSM for representing the contextual view of a system is the ER approach (Reference 14). Four terms are used in the ER approach: entities, relationships, attributes, and value sets. Figure 9 is an example of an ER diagram that is a useful visual aid when a small number of objects are being displayed.

Entities are identifiable objects in the environment. Some examples are a momentum wheel, a user, a CRT display, a fuel tank, Earth, and a spacecraft. Events (e.g., start of maneuver, end of integration step) are considered to be entities in the ER approach. In the CSM, the entities that correspond to events can be identified separately but share all of the properties of entities. In the following discussion, entities may includes events.

Relationships are associations among entities and are defined as are relations in discrete mathematics (Reference 15). Examples of relations in Figure 9 are T/S for thruster-spacecraft and F/T/S for fuel-thruster-spacecraft.

Information about entities and relationships is expressed by a set by attribute-value pairs. An attribute is a property or feature of the entity or relationship. For example, the entity "fuel tank" have an attribute of volume.

Value sets combine the concepts of the units of measure with ranges and types of acceptable values for attributes. Figure 9 shows the value set for the attribute "center of gravity."

A valuable conceptual feature of the ER approach is the ability to associate attributes with relationships as well as entities. The attribute, thruster position, in Figure 9 is properly associated with the thruster-spacecraft relation. It would be inaccurate to associate it with either of the entities "thruster" or "spacecraft" alone.

Figure 10 shows an example of the CSM's dynamic view, representing the behavior of the system over time. The notation used is the state transition diagram, a directed graph in which the nodes correspond to states of the system and the directed arcs show the possible changes in state. Events in the environment (e.g., a user selects a menu option) provide the stimuli to trigger a state change.

STEP 2: THE DEFINITION OF MEASURES

As the second step in the revised procedure (Figure 11), 58 measures were defined using the CSM as a basis. Because of the CSM's graphical style, there existed many opportunities to use basic counts of the objects in the diagrams. From the functional view, some obvious explicit measures were counts of the constituents of data flow diagrams: functional primitives, data flows, data stores, and external entities. From the contextual view, the explicit measures were counts of entities, events, relations, value sets, and attributes. The dynamic view generated counts of states and transitions.

To these explicit counts were added a host of analytic measures. Some were derived from applying various normalization factors to the explicit counts to obtain measures like arc weight or relation density. Other analytic measures were based on suggestions of other investigators, for example, weighted function and derivation set complexity. The complete definitions of all 58 measures are given in Reference 7.

STEP 3: THE CSM APPLICATION

Step 3 (Figure 12) involved applying the CSM to a real system. The selected system was the Yaw Maneuver Control Utility (YMCU) of the Earth Radiation Budget Satellite (ERBS). Although identified as a utility, the system was not a trivial one. It consisted of 85 modules comprising 11,200 delivered source lines of FORTRAN.

The requirements for YMCU were recast in the form of the CSM, producing a new document (Reference 16).

STEP 4: THE EXTRACTION OF MEASURES

Using the CSM representation, the recommended measures were extracted as Step 4 in the revised approach (Figure 13). Some of the extracted metric values are shown in Figure 13, organized according to the three views of the CSM. Details of the matrices extraction procedure are found in Reference 17.

STEP 5: THE ASSESSMENT OF THE EXPERIMENT

Step 5 (Figure 14) required assessing the process and the resulting measures. The process was demonstrated to be feasible through the experiment of extracting the measures from the YMCU. A consequence of the process was the production of a recast requirements document using the CSM. The CSM

version appeared to be clearer, more accessible, and more informative through its nonnarrative style featuring diagrams, lists, and tables. We will want to obtain many comments from users of requirements documents to determine if this optimistic assessment is justified.

A characteristic of the metrics extraction experiment from the outset was the collection of effort data in an attempt to understand the cost of obtaining the CSM representation. The data revealed that 1.7 staff months were spent constructing the CSM representation of the YMCU. Standard SEL effort data on the YMCU software development project showed that 2.1 staff months were charged to traditional requirements analysis. Many factors should be covered in a thorough discussion of the relative effort required to build the CSM representation. Without reproducing that discussion, which is pursued in References 7 and 17, one conclusion is clear: both effort figures are of the same order of magnitude. That is, 10 or 20 times more effort was not required to build the CSM model when compared to traditional requirements analysis. If the CSM is clear and more understandable as it seems to be, the effort in successive phases may be reduced.

Two observations are clear regarding the assessment of measures. First, the collection of measures constitutes only a single datum in any attempt to draw inferences from the measures. More projects would need to be measured before any patterns might begin to emerge. One superficial relationship stands out. The 39 functional primitives are approximately one half of the number of modules (85) in the delivered product. Whether any such relation persists is open to speculation.

The second observation is that human judgment continues to play a role in these specification measures. Our preference for objective measures is no assurance that we have eliminated subjective considerations. The superficial relation noted above provides a ready example. The identification of functional primitives is sensitive to the procedure for decomposing processes in data flow analysis. At least four guidelines exist in DeMarco's books alone for determining when functional decomposition should be ceased (References 10 and 12).

Although the CSM has not eliminated the subjective component in specification measurement, it has, we believe, reduced its effect dramatically when compared to measures drawn from narrative statements of requirements. The enumerative style

of the CSM and its reliance on notations that have some internal consistency give us reason to believe the CSM has made progress toward objective specification measurement.

CONCLUSIONS

This investigation has confirmed that objective specification measures need a disciplined representation of requirements (Figure 15). The CSM has been advanced as a framework for capturing software requirements. The CSM fulfills its original purpose by enabling the definition of objective specification measures. Its multiple views are more revealing than any single perspective on requirements. The CSM is a new product of the software development process, available early, and therefore able to assist later stages. The nonnarrative CSM style affords visibility at a life cycle phase in which the identification of configuration control items is extremely difficult through traditional means. The CSM, being more accessible and modular, facilitates reusability. Other investigators have recognized the benefits of achieving reusability during the earliest life cycle phases (References 18 and 19).

By representing the context of the software system, the CSM is capturing valuable information. The environment of the system is the starting point for object-oriented design.

The goal of this study was portrayed in Figure 3 as developing a measurement shell that would encase the requirements and supply measures of the properties of interest. Through the CSM representation, measures have been defined and extracted. These measures serve as early indicators of properties like size and complexity. For other properties, although no direct measures were defined, the CSM representation will make it easier to detect, for example, inconsistency and incompleteness.

This study has contributed to our understanding of the role that specification measures might fulfill in the flight dynamics environment. We intend to consider the CSM and its derived measures for application on new software projects.

ACKNOWLEDGMENTS

This study of specification measures has benefited from the comments and suggestions of many SEL colleagues, especially V. Church, F. McGarry, D. Card, L. Jordan, W. Decker, and V. Basili.

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. W. W. Agresti, F. E. McGarry, D. N. Card, et al., "Measuring Software Technology," Program Transformation and Programming Environments. New York: Springer-Verlog, 1984

3. V. R. Basili and K. Freburger, "Programming Measurement and Estimation in the Software Engineering Laboratory," Journal of Systems and Software, February 1981, vol. 2, no. 1

4. Software Engineering Laboratory, 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

5. R. S. Pressman, Software Engineering: A Practitioner's Approach. New York: McGraw-Hill, 1982

6. W. W. Agresti, "Measuring Program Maintainability," Journal of Systems Management, vol. 33, no. 3, 1982

7. Software Engineering Laboratory, SEL-84-003, Investigation of Specification Measures for the Software Engineering Laboratory, W. W. Agresti, V. Church, and F. E. McGarry, December 1984

8. B. W. Boehm, Software Engineering Economics. Englewood Cliffs, N.J.: Prentice-Hall, 1981

9. Software Engineering Laboratory, SEL-78-006, GSFC Software Engineering Research Requirements Analysis Study, P. Scheffer and T. Velez, November 1978

10. T. DeMarco, Controlling Software Projects. New York: Yourdon Press, 1982

11. R. T. Yeh and P. Zave, "Specifying Software Requirements, "Proceedings of the IEEE, vol. 68, no. 9, 1980

12. T. DeMarco, Structured Analysis and System Specification. New York: Yourdon, Inc., 1978

13. H. Simon, The Sciences of the Artificial. Cambridge, Mass.: M.I.T. Press, 1970

14. P. Chen, "The Entity-Relationship Model--Toward a Unified View of Data," ACM Transactions on Data Base Systems, March 1976

15. C. L. Liu, Elements of Discrete Mathematics. New York: McGraw-Hill, 1977

16. Computer Sciences Corporation, Informational Memorandum, "Case Study in Recasting Flight Dynamics Software Requirements Using the Composite Specification Model (CSM)," W. Agresti, December 1984

17. --, Informational Memorandum, "Extracting Specification Measures From Flight Dynamics Software Requirements," W. Agresti, December 1984

18. Y. Matsumoto and K. Matsumura, "A Specification Analysis and Documentation System for Process Control Software," Proceedings, IEEE COMPSAC, 1981

19. S. J. Greenspan, J. Mylopoulos, and A. Borgida, "Capturing More World Knowledge in the Requirements Specification," Proceedings, Sixth International Conference on Software Engineering, New York: Computer Societies Press, 1982

THE VIEWGRAPH MATERIALS

for the

W. AGRESTI PRESENTATION FOLLOW

AN APPROACH TO DEVELOPING SPECIFICATION MEASURES

W. AGRESTI

714-AGR-(89)

Figure 1

WHO USES REQUIREMENTS?

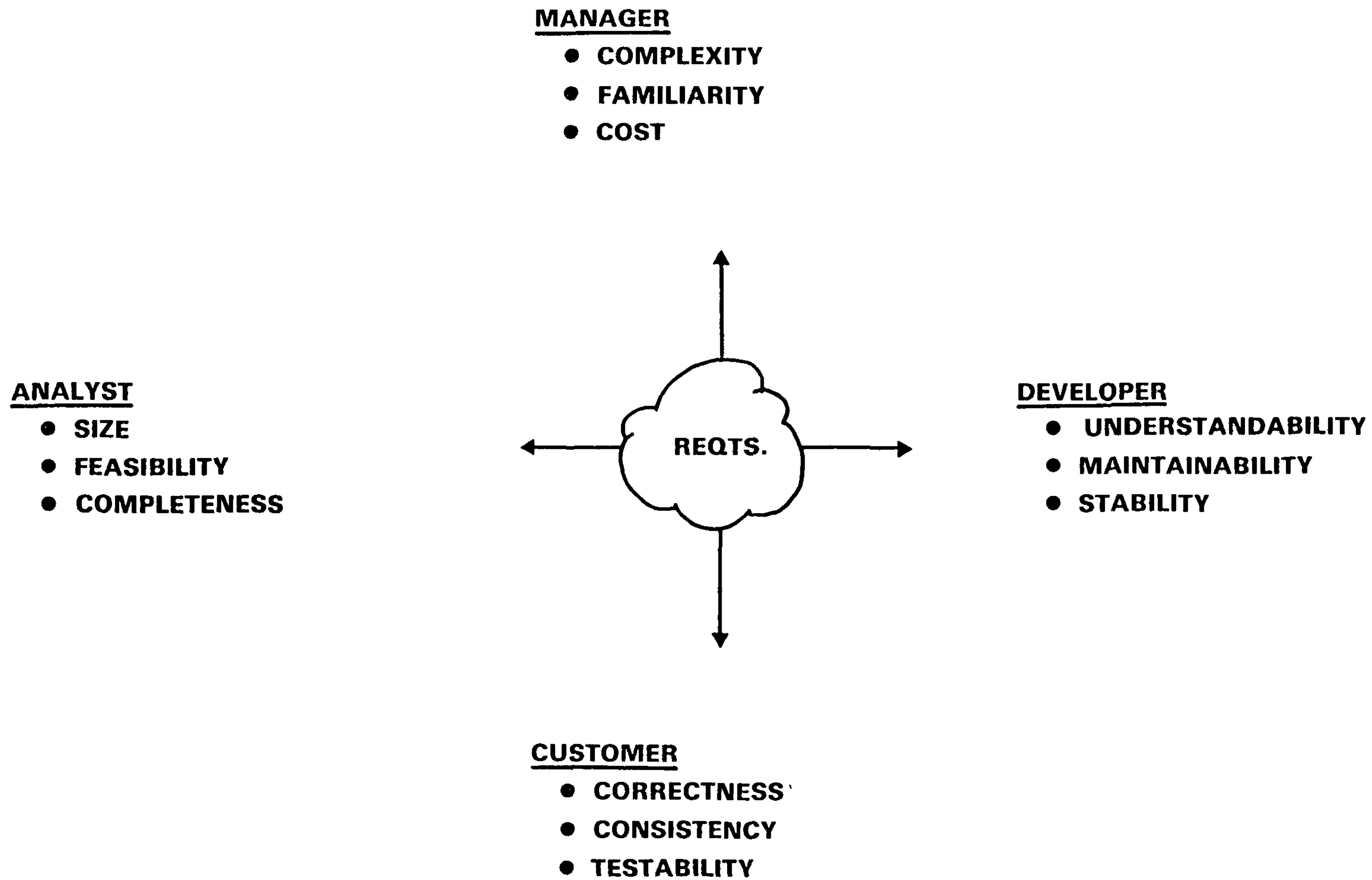


Figure 2

WHO USES REQUIREMENTS?

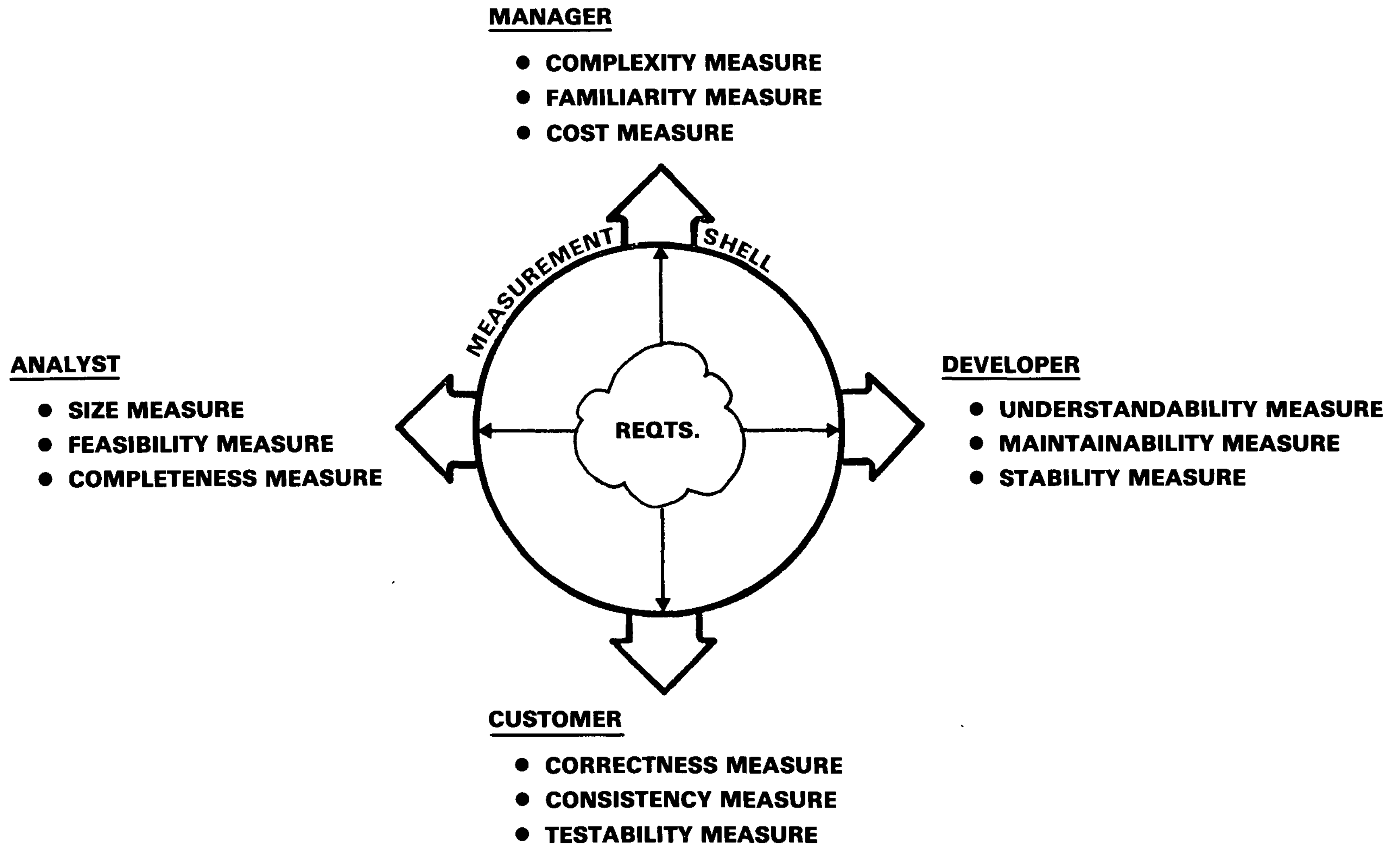


Figure 3

OUR APPROACH

FOCUS: OBJECTIVE MEASURES

PROCEDURE: DEFINED 29 EXPLICIT MEASURES BASED ON EXISTING REQUIREMENTS SPECIFICATIONS

NUMBER OF PAGES

NUMBER OF CONSTRAINTS

NUMBER OF I/O REQUIREMENTS

•

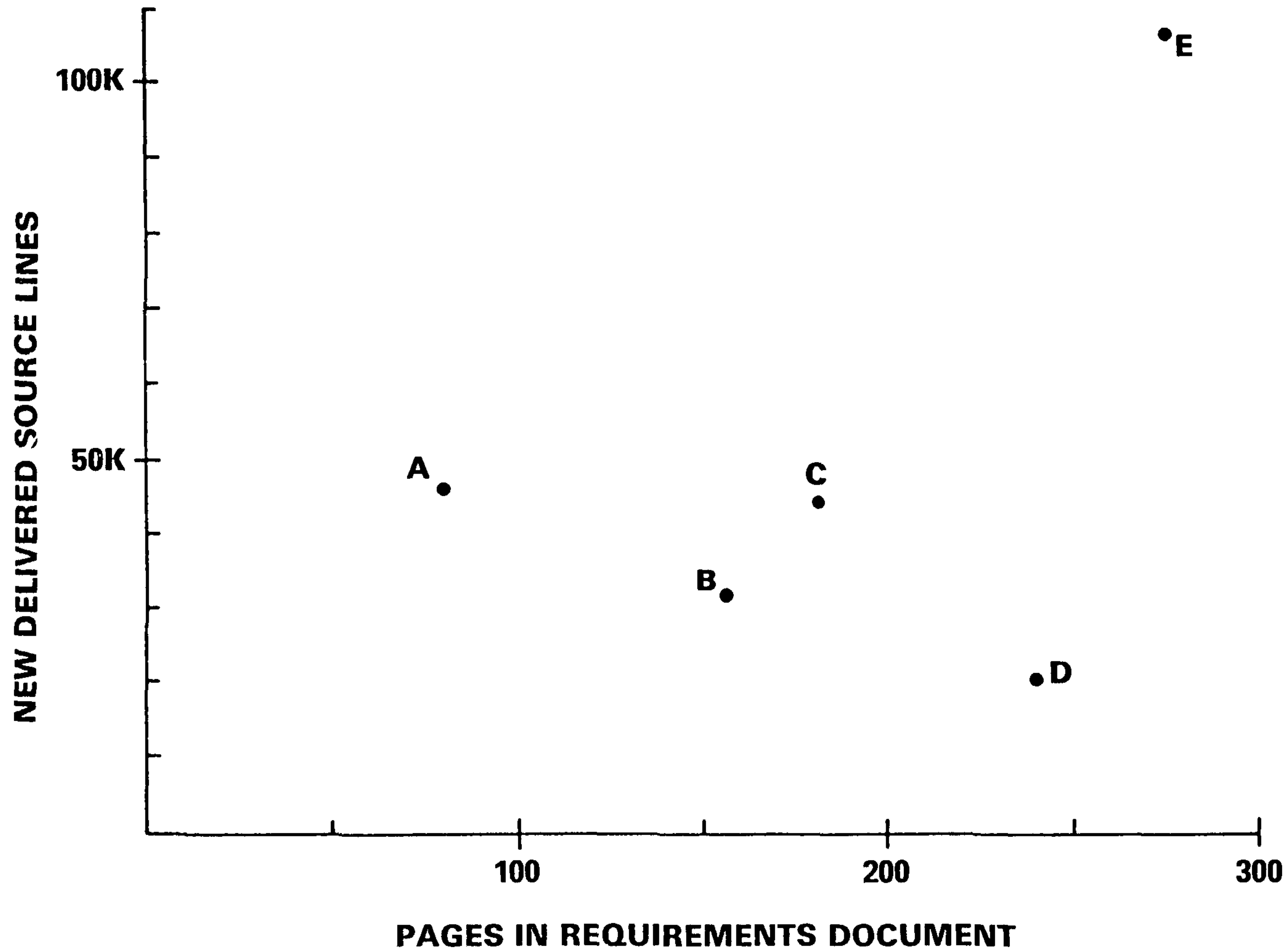
•

•

RESULT: MEASURES WERE EXTRACTABLE BUT NOT USEFUL

Figure 4

FIVE FLIGHT DYNAMICS SOFTWARE PROJECTS NEW SOURCE LINES VS. PAGES OF REQUIREMENTS



LESSON: TO DEVELOP OBJECTIVE SPECIFICATION MEASURES, REPRESENTATION IS EVERYTHING!

Figure 5

OUR REVISED APPROACH

STEP 1: PROPOSE A NEW REPRESENTATION

STEP 2: DEFINE SPECIFICATION MEASURES BASED ON IT

STEP 3: APPLY IT TO A REAL SYSTEM

STEP 4: EXTRACT THE MEASURES

STEP 5: ASSESS THE PROCESS AND THE RESULTING MEASURES

Figure 6

STEP 1:
PROPOSE A NEW REPRESENTATION

COMPOSITE SPECIFICATION MODEL (CSM)

RATIONALE: REQUIREMENTS FOR COMPLEX SOFTWARE NEED TO BE SPECIFIED FROM MULTIPLE VIEWPOINTS

VIEWPOINT	NOTATION
• FUNCTIONAL	• DATA FLOW
• CONTEXTUAL	• ENTITY/RELATIONSHIP
• DYNAMIC	• STATE/TRANSITION

Figure 7

EXAMPLE OF FUNCTIONAL VIEW

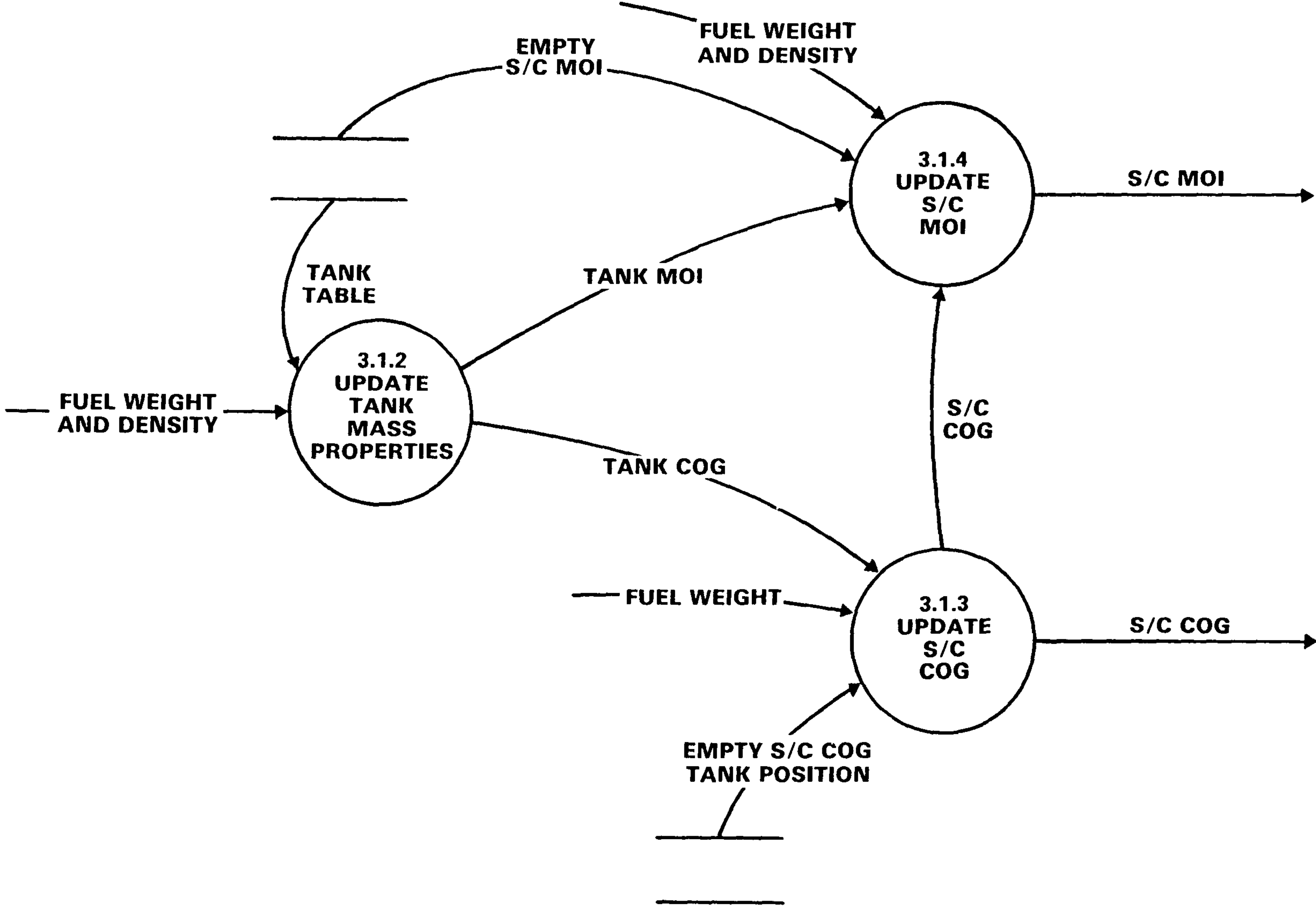
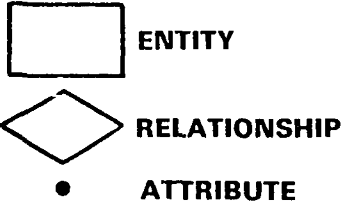


Figure 8

EXAMPLE OF CONTEXTUAL VIEW



SPACECRAFT
- MASS
- CENTER OF GRAVITY (EMPTY)

VALUE SET	
TYPE:	REAL
STRUCTURE:	VECTOR
SIZE:	3
RANGE:	–, 0, +
COORDINATE SYSTEM:	S/C BODY
UNITS:	METERS

F/T/S
- THRUST TORQUE VECTOR

T/S
- THRUSTER POSITION
- THRUST DIRECTION

FUEL
- DENSITY

F/T
- COOLING CURVE
- FUEL FLOW RATE

THRUSTER

Figure 9

EXAMPLE OF DYNAMIC VIEW (STATES AND TRANSITIONS)

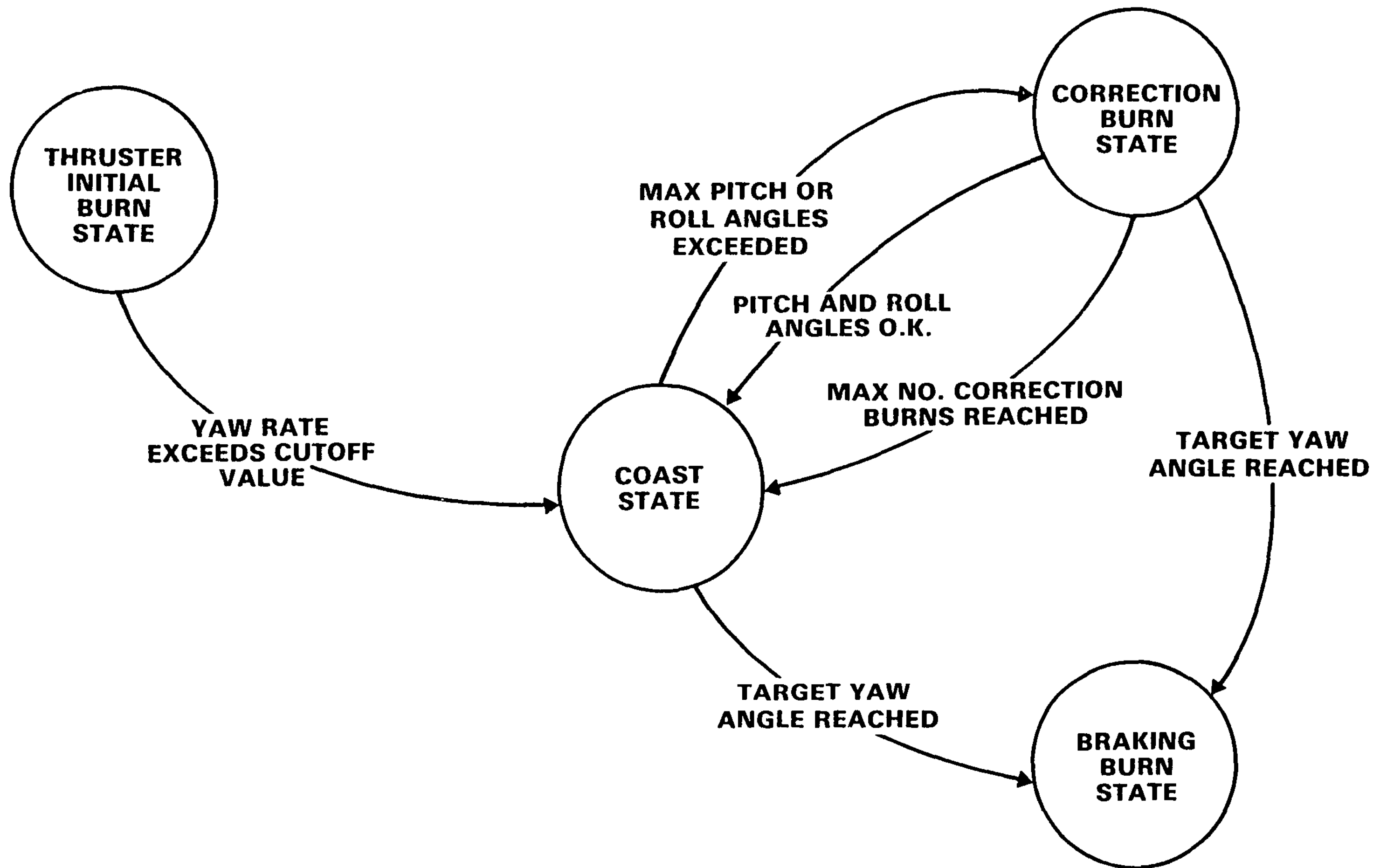


Figure 10

STEP 2: DEFINE MEASURES BASED ON THE COMPOSITE SPECIFICATION MODEL

58 MEASURES DEFINED

EXPLICIT	ANALYTIC
NUMBER OF FUNCTIONAL PRIMITIVES	WEIGHTED FUNCTION
NUMBER OF DATA ITEMS	RELATION DENSITY
NUMBER OF STATES	ARC WEIGHT
•	•
•	•
•	•

Figure 11

STEP 3: APPLY THE COMPOSITE SPECIFICATION MODEL TO A REAL SYSTEM

- YAW MANEUVER CONTROL UTILITY OF EARTH RADIATION BUDGET SATELLITE (ERBS)
- FORTRAN
- 11,200 DELIVERED SOURCE LINES
- 85 MODULES

Figure 12

STEP 4 EXTRACT THE MEASURES

MEASURE	VALUE
FUNCTIONAL VIEW	
• FUNCTIONAL PRIMITIVES	39
• INTERFACE COUNT	3
• INTERNAL ARCS	60
• INTERNAL DATA ITEMS	42
• SYSTEM IN/OUT DATA ITEMS	67
• FILE IN/OUT DATA ITEMS	74
• WEIGHTED FUNCTION	688
CONTEXTUAL VIEW	
• ENTITIES	11
• EVENTS	14
• RELATIONS	19
• ATTRIBUTES	91
• VALUE SETS	29
DYNAMIC VIEW	
• STATES	7
• TRANSITIONS	11

Figure 13

STEP 5: ASSESS THE PROCESS AND RESULTING MEASURES

PROCESS

- EFFORT REQUIRED FOR CSM MAY REDUCE EFFORT IN LATER PHASES
  - 2.1 STAFF MONTHS FOR TRADITIONAL REQUIREMENTS ANALYSIS
  - 1.7 STAFF MONTHS FOR BUILDING CSM

RESULTING MEASURES

- HUMAN JUDGMENT STILL IS A FACTOR
- NEED TO MEASURE MORE PROJECTS

Figure 14

CONCLUSIONS

- **OBJECTIVE SPECIFICATION MEASURES NEED DISCIPLINED REPRESENTATION OF REQUIREMENTS**

- **BUILDING THE CSM IS FEASIBLE**
  - **YIELDS OBJECTIVE SPECIFICATION MEASURES**
  - **MULTIPLE VIEWS ARE MORE REVEALING**
  - **MORE EFFECTIVE REPRESENTATION TO BEGIN DESIGN**

- **CAPTURING THE CONTEXT OF A SYSTEM IS BENEFICIAL**
  - **SOURCE OF CHANGES TO THE SYSTEM**
  - **LOGICAL PREDECESSOR OF OBJECT-ORIENTED DESIGN**

Figure 15

N86-19969

Evaluating Software Testing Strategies

Richard W. Selby, Jr. and Victor R. Basili
Department of Computer Science
University of Maryland
College Park, Maryland 20742
(301) 454-4247

Jerry Page
Computer Sciences Corp., Silver Spring, MD

Frank E. McGarry
NASA/GSFC, Greenbelt, MD

ABSTRACT

This study compares the strategies of code reading, functional testing, and structural testing in three aspects of software testing: fault detection effectiveness, fault detection cost, and classes of faults detected. Thirty two professional programmers applied the three techniques to three unit-sized programs in a fractional factorial experimental design. The major results of this study so far are the following. 1) Code readers detected more faults than did those using the other techniques, while functional testers detected more faults than did structural testers. 2) Code readers had a higher fault detection rate than did those using the other methods, while there was no difference between functional testers and structural testers. 3) Subjects testing the abstract data type detected the most faults and had the highest fault detection rate, while individuals testing the database maintainer found the fewest faults and spent the most effort testing. 4) Subjects of intermediate and junior expertise were not different in number or percentage of faults found, fault detection rate, or fault detection effort; subjects of advanced expertise found a greater number of faults than did the others, found a greater percentage of faults than did just those of junior expertise, and were not different from the others in either fault detection rate or effort. 5) Code readers and functional testers both detected more omission faults and more control faults than did structural testers, while code readers detected more interface faults than did those using the other methods.

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.

1. Introduction

The processes of software testing and defect detection continue to challenge the software community. Even though the software testing and defect detection activities are inexact and inadequately understood, they are crucial to the success of a software project. The controlled study presented addresses the uncertainty of how to test software effectively. In this investigation, common testing techniques were applied to different types of software by a representative group of programming professionals. This work is intended to characterize how testing effectiveness relates to several factors: testing technique, software type, fault type, tester experience, and any interactions among these factors.

This paper gives an overview of the testing techniques examined, investigation goals, experimental design, and data analysis. The results presented are from a preliminary analysis of the data; a more complete analysis appears elsewhere [Selby 84, Basili & Selby 85].

2. Testing Techniques

To demonstrate that a particular program actually meets its specifications, professional software developers currently utilize many different testing methods. In functional testing, which is a "black box" approach [Howden 80], a programmer constructs test data from the program's specification through methods such as equivalence partitioning and boundary value analysis [Myers 79]. The programmer then executes the program and contrasts its actual behavior with that indicated in the specification. In structural testing, which is a "white box" approach [Howden 78, Howden 81], a programmer inspects the source code and then devises and executes test cases based on the percentage of the program's statements or expressions executed (the "test set coverage") [Stucki 77]. The structural coverage criteria used in this study was 100% statement coverage. In code reading by stepwise abstraction, a person identifies prime subprograms in the software, determines their functions, and composes these functions to determine a function for the entire program [Mills 72, Linger, Mills & Witt 79]. The code reader then compares this derived function and the specifications (the intended function).

2.1. Investigation Goals

The goals for this study are to compare the three common testing techniques of code reading, functional testing, and structural testing in terms of 1) fault detection effectiveness, 2) fault detection cost, and 3) classes of faults detected. An example research question in each of these goal areas is as follows. Which testing technique (code reading, functional testing, or structural testing) leads to the detection of the most faults? Which testing technique leads to the highest fault detection rate (#faults/effort)? Which testing techniques capture which classes of faults?

3. Empirical Study

Admittedly, the goals for this study are quite ambitious. In no way is it implied that this study can definitively answer all of these questions for all environments. It is

intended, however, that the statistically significant analysis undertaken lends insights into their answers and into the merit and appropriateness of each of the techniques.

A primary consideration in this study was to use a realistic testing environment to assess the effectiveness of these different testing strategies, as opposed to creating a best possible testing situation [Hetzel 76]. Thus, 1) the subjects chosen for the study were professional programmers with a wide range of experience, 2) the programs tested correspond to different types of software and reflect common programming style, and 3) the faults in the programs were representative of those frequently occurring in software. Sampling the subjects, programs, and faults in this manner is intended to provide a reasonable evaluation of the testing methods, and to facilitate the generalization of the results to other environments. Note that prior to this experiment, we conducted a similar testing study involving 42 advanced students from the University of Maryland [Basili & Selby 85].

The following sections describe the empirical study undertaken, including the selection of subjects, programs, and experimental design, and the operation of the study.

3.1. Subjects

The 32 subjects in the study were programming professionals from NASA and Computer Sciences Corporation. These individuals were mathematicians, physicists, and engineers that developed ground support software for satellites. They had familiarity with all three testing techniques, but used functional testing primarily. R. W. Selby conducted a three hour tutorial on the testing techniques for the subjects. The subjects were selected to be representative of three different levels of computer science expertise: advanced, intermediate, and junior. Several criteria were considered in the association of a subject with an expertise level, including years of professional experience, degree background, and their manager's suggested assignment. The individuals examined included eight advanced, eleven intermediate, and thirteen junior subjects; these groups had an average of 15.0, 10.9, and 6.1 years of professional experience, respectively, with an overall average of 10.0 (SD = 5.7) years.

3.2. Programs

The three FORTRAN programs used in the investigation were chosen to be representative of several different software types: a text formatter, a numeric abstract data type, and a database maintainer. The programs are summarized in Figure 1. The specifications for the programs and their source code appear in [Selby 84].

Figure 1. The programs tested.					
program	source lines	executable statments	cyclomatic complexity	#routines	#faults
text formatter	169	55	18	3	9
numeric data abstraction	147	48	18	9	7
database maintainer	365	144	57	7	12

There exists some differentiation in size among the programs, and they are a realistic size for unit testing. The first program is a text formatting program, which also appeared in [Myers 78]. A version of this program, originally written by [Naur 69] using techniques of program correctness proofs, was analyzed in [Goodenough & Gerhart 75]. The second program is a numeric data abstraction consisting of a set of list processing utilities. This program was submitted for a class project by a member of an intermediate level programming course at the University of Maryland [McMullin & Gannon 80]. The third program is a maintainer for a database of bibliographic references. This program was analyzed in [Hetzel 76], and was written by a systems programmer at the University of North Carolina Computation Center.

3.3. Faults

The 28 faults in the programs comprise a reasonable distribution of faults that commonly occur in software [Basili & Weiss 82, Basili & Perricone 84]. All the faults in the database maintainer and the numeric abstract data type were made during the actual development of the programs. The text formatter contains a mix of faults made by the original programmer and faults seeded in the code. Note that this investigation involves only those types of faults occurring in the source code, not other types such as those in the requirements or specifications.

Two abstract classification schemes characterize the faults in the programs. One fault categorization method separates faults of omission from faults of commission. A second fault categorization scheme partitions software faults into the six classes of 1) initialization, 2) computation, 3) control, 4) interface, 5) data, and 6) cosmetic. An explanation of these classification schemes appeared in [Basili & Perricone 84], and the faults themselves are described in [Selby 84]. These two classification schemes are intended to distinguish among different reasons that programmers make faults in software development. The consistent application of the two schemes to the faults in the programs resulted in a mutually exclusive and exhaustive categorization; it is certainly possible that another analyst could have a different interpretation (see Figure 2).

Figure 2. Distribution of faults in the programs.			
	Omission	Commission	Total
Initialization	0	2	2
Computation	2	2	4
Control	2	4	6
Interface	2	11	13
Data	2	0	2
Cosmetic	0	1	1
Total	8	20	28

3.4. Experimental Design

The experimental design applied was a fractional factorial design [Cochran & Cox 50, Box, Hunter, & Hunter 78]. All of the subjects tested each of the three programs and used each of the three techniques. Of course, no one tested a given program more than once. The order of presentation of the testing techniques was randomized among the subjects in each level of expertise. A factorial analysis of variance (ANOVA) model supports the analysis of both the main effects (testing technique, software type, programmer expertise) and any interactions among the main effects.

The subjects examined in the study were random samples of programmers from the large population of programmers at each of the levels of expertise. If the samples examined are truly representative of the population of programmers at each expertise level, the inferences from the analysis can then be generalized across the whole population of individuals at each expertise level, not just across the particular subjects in the sample chosen.

3.5. Experimental Operation

The controlled study included five phases: training, three testing sessions, and a follow-up session. All groups of subjects were exposed to a similar amount of training on the testing techniques before the study began. In the testing sessions, the individuals were requested to use the testing techniques to the best of their ability. The subjects' desire for the study's outcome to improve their software testing environment ensured reasonable effort on their part. Note that when the subjects were applying either functional or structural testing, they generated and executed their own test data; no test data sets were provided. At the end of each of the testing sessions, the subjects estimated the amount of time spent detecting faults and the percentage of the faults in the program that they thought were uncovered. The study concluded with a debriefing session for discussing the preliminary results and the subjects' observations.

4. Data Analysis

This section presents the data analysis according to the three goal areas discussed earlier.

4.1. Fault Detection Effectiveness

The first goal area examines the factors contributing to fault detection effectiveness. The following sections present the relationship of fault detection effectiveness to testing technique, software type, programmer expertise, and self-estimate of faults detected.

4.1.1. Testing Technique

The subjects applying code reading detected an average of 5.09 (SD = 1.92) faults per program, persons using functional testing found 4.47 (SD = 1.34), and those applying structural testing uncovered 3.25 (SD = 1.80); the subjects detected an overall average of 4.27 (SD = 1.86) faults per program. Subjects using code reading detected 1.24 more faults per program than did subjects using either functional or structural testing ($\alpha < .0001$, 95% c.i. 0.73 – 1.75).[1] Subjects using functional testing detected 1.11 more faults per program than did those using structural testing ($\alpha < .0007$, 95% c.i. 0.52 – 1.70). Since the programs each had a different number of faults, an alternate interpretation compares the percentage of the programs' faults detected by the techniques. The techniques performed in the same order when percentages are compared: subjects applying code reading detected 16.0% more faults per program than did subjects using the other techniques ($\alpha < .0001$, c.i. 9.9 – 22.1%), and subjects applying functional testing detected 11.2% more faults than did those using structural testing ($\alpha < .003$, c.i. 4.1 – 18.3%). Thus comparing either the number or percentage of faults detected, individuals using code reading observed the most faults, persons applying functional testing found the second most, and those doing structural testing uncovered the fewest.[2]

4.1.2. Software Type

The subjects testing the abstract data type detected an average of 5.22 (SD = 1.75) faults, persons testing the text formatter found 4.19 (SD = 1.73), and those testing the database maintainer uncovered 3.41 (SD = 1.66). The application of Tukey's multiple comparison reveals that subjects detected the most faults in the abstract data type, the second most in the text formatter, and the fewest faults in the database maintainer (simultaneous $\alpha < .05$). This ordering is the same for both number and percentage of faults detected.

4.1.3. Programmer Expertise

Subjects of advanced expertise detected an average of 5.00 (SD = 1.53) faults, persons of intermediate expertise found 4.18 (SD = 1.99), and those of junior expertise uncovered 3.90 (SD = 1.83). Subjects of intermediate and junior expertise were not statistically different in terms of either number or percentage of faults observed ($\alpha > .05$).

---

[1] The probably of Type I error is reported, the probability of erroneously rejecting the null hypothesis. The abbreviation "c.i." stands for confidence interval. The intervals reported are all 95% confidence intervals.

[2] Recall that the individuals used the following techniques: code reading by stepwise abstraction, functional testing using equivalence partitioning and boundary value analysis, and structural testing with 100% statement coverage criteria.

individuals of advanced expertise detected both a greater number and percentage of faults than did those of junior expertise ($\alpha < .05$). Persons of advanced expertise detected a greater number of faults that did those of intermediate expertise ($\alpha < .05$), but the advanced and intermediate groups were not statistically different in percentage of faults detected ($\alpha > .05$).

4.1.4. Self-Estimate of Faults Detected

At the completion of a testing session, the subjects estimated the percentage of a program's faults they thought they had uncovered. This estimation of the number of faults uncovered correlated reasonably well with the actual percentage of faults detected ($R = .57$, $\alpha < .0001$). Further investigation shows that individuals using certain techniques gave better estimates: code readers gave the best estimates (Pearson $R = .79$, $\alpha < .0001$), structural testers gave the second best estimates ($R = .57$, $\alpha < .0007$), and functional testers gave the worst estimates (no correlation, $\alpha > .05$). This observation suggests that the code readers were more certain of the effectiveness they had in revealing faults in the programs.

4.2. Fault Detection Cost

The second goal area examines the factors contributing to fault detection cost. The following sections present the relationship of fault detection cost to testing technique, software type, and programmer expertise.

4.2.1. Testing Technique

The subjects applying code reading detected faults at an average rate of 3.33 (SD = 3.42) faults per hour, persons using functional testing found faults at 1.84 (SD = 1.06) faults per hour, and those applying structural testing uncovered faults at a rate of 1.82 (SD = 1.24) faults per hour; the subjects detected faults at an overall average rate of 2.33 (SD = 2.28) faults per hour. Subjects using code reading detected 1.49 more faults per hour than did subjects using either functional or structural testing ($\alpha < .0003$, c.i. 0.75 – 2.23). Subjects using functional and structural testing were not statistically different in fault detection rate ($\alpha > .05$). The subjects spent an average of 2.75 (SD = 1.57) hours per program detecting faults. Comparing the total time spent in fault detection, the techniques were not statistically different ($\alpha > .05$). Thus, subjects using code reading detected faults at a higher rate than did those applying functional or structural testing, while the total fault detection effort was not different among the methods.

4.2.2. Software Type

The subjects testing the abstract data type detected faults at an average rate of 3.70 (SD = 3.26) faults per hour, persons testing the text formatter found faults at 2.15 (SD = 1.10) faults per hour, and those testing the database maintainer uncovered faults at a rate of 1.14 (SD = 0.79) faults per hour. Applying Tukey's multiple comparisons, the fault detection rate was higher in the abstract data type than it was for either the text formatter or the database maintainer, while the text formatter and the database maintainer were not statistically different (simultaneous $\alpha < .05$). The overall time spent in fault detection also differed among the programs. Subjects spent more time testing

the database maintainer than they spent on either the text formatter or the abstract data type, while the time spent on the text formatter and the abstract data type was not statistically different (simultaneous $\alpha < .05$). Thus, subjects uncovered faults at the fastest rate in the abstract data type, and spent the most time testing the database maintainer.

4.2.3. Programmer Expertise

Subjects of advanced expertise detected faults at an average rate of 2.36 (SD = 1.61) faults per hour, subjects of intermediate expertise found faults at 2.53 (SD = 2.48) faults per hour, and subjects of junior expertise uncovered faults at a rate of 2.14 (SD = 2.48) faults per hour. Programmer expertise level had no relation to either fault detection rate or total effort spent in fault detection (both $\alpha > .05$).

4.3. Characterization of Faults Detected

The third goal area focuses on determining what classes of faults are detected by the different techniques. An earlier section characterized the faults in the programs by two different classification schemes: omission or commission, and initialization, control, data, computation, interface, or cosmetic.

When the faults are partitioned according to the omission/commission scheme, a distinction surfaces among the techniques. Subjects using either code reading or functional testing observed more omission faults than did individuals applying structural testing, while there was no difference between code reading and functional testing. Since a fault of omission occurs as a result of some segment of code being left out ("omitted"), you would not expect structurally generated test data to find such a fault.

Dividing the faults according to the second fault classification scheme reveals a few distinctions among the methods. Subjects using code reading detected more interface faults than did those applying either of the other methods, while there was no difference between functional and structural testing. This suggests that code reading by abstracting and composing program functions across modules must be an effective technique for finding interface faults. Individuals using either code reading or functional testing detected more control faults than did persons applying structural testing. Recall that subjects applying structural testing determined the execution paths in a program and then generated test data that executed 100% of the program's statements. One would expect that more control path faults would be found by such an approach. However, structural testing did not do as well as the others in this fault class, suggesting the inadequacy of statement coverage criteria.

5. Preliminary Conclusions

This study compares the strategies of code reading, functional testing, and structural testing in three different aspects of software testing: fault detection effectiveness, fault detection cost, and classes of faults detected. Each of the three testing techniques showed merit in this evaluation. The investigation was intended to compare the different testing strategies in a representative testing situation, using professional programmers, different software types, and common software faults.

The major results of this study so far are the following. 1) Code readers detected more faults than did those using the other techniques, while functional testers detected more faults than did structural testers. 2) Code readers had a higher fault detection rate than did those using the other methods, while there was no difference between functional testers and structural testers. 3) Subjects testing the abstract data type detected the most faults and had the highest fault detection rate, while individuals testing the database maintainer found the fewest faults and spent the most effort testing. 4) Subjects of intermediate and junior expertise were not different in number or percentage of faults found, fault detection rate, or fault detection effort; subjects of advanced expertise found a greater number of faults than did the others, found a greater percentage of faults than did just those of junior expertise, and were not different from the others in either fault detection rate or effort. 5) Code readers and functional testers both detected more omission faults and more control faults than did structural testers, while code readers detected more interface faults than did those using the other methods.

A comparison of professional programmers using code reading with novice and junior programmers using the technique suggests a possible learning curve. In a testing study similar to this one, using a group of advanced students, code readers and functional testers were equally effective in fault detection while structural testers were either equally effective or inferior [Basili & Selby 85]. Also, the three techniques were not different in fault detection rate. Further comparison of this study with other testing studies, including [Hetzel 76, Myers 78, Hwang 81], appears in [Basili & Selby 85].

Investigations related to this work include studies of fault classification [Basili & Weiss 82, Johnson, Draper & Soloway 83, Ostrand & Weyuker 83, Basili & Perricone 84] and Cleanroom software development [Selby, Basili & Baker 84]. In the Cleanroom software development approach, techniques such as code reading are used in the development of software completely off-line. In the above study, systems developed using Cleanroom met system requirements more completely and had a higher percentage of successful operational test cases than did systems developed with a more traditional approach.

This empirical study is intended to advance the understanding of how various software testing strategies contribute to the software development process and to one another. The results given were calculated from a set of individuals applying the three techniques to unit-sized programs – the direct extrapolation of the findings to other testing environments is not implied. However, valuable insights have been gained and additional areas of analysis and interpretation appear in [Selby 84, Basili & Selby 85].

6. Acknowledgement

The authors are grateful to the subjects from Computer Sciences Corporation and NASA Goddard for their enthusiastic participation in this study.

7. References

[Basili & Weiss 82]
V. R. Basili and D. M. Weiss, Evaluating Software Development by Analysis of Changes: The Data from the Software Engineering Laboratory*, Dept. Com. Sci., Univ. Maryland, College Park, Tech. Rep. TR-1236, Dec. 1982..

[Basili & Perricone 84]
V. R. Basili and B. T. Perricone, Software Errors and Complexity: An Empirical Investigation, *Communications of the ACM* **27**, 1, pp. 42-52, Jan. 1984.

[Basili & Selby 85]
V. R. Basili and R. W. Selby, Jr., Comparing the Effectiveness of Software Testing Strategies, Dept. Com. Sci., Univ. Maryland, College Park, Tech. Rep., 1985.

[Box, Hunter, & Hunter 78]
G. E. P. Box, W. G. Hunter, and J. S. Hunter, *Statistics for Experimenters,* John Wiley & Sons, New York, 1978.

[Cochran & Cox 50]
W. G. Cochran and G. M. Cox, *Experimental Designs,* John Wiley & Sons, New York, 1950.

[Goodenough & Gerhart 75]
J. B. Goodenough and S. L. Gerhart, Toward a Theory of Test Data Selection, *IEEE Trans. Software Engr.*, pp. 156-173, June 1975.

[Hetzel 76]
W. C. Hetzel, An Expermental Analysis of Program Verification Methods, Ph.D. Thesis, Univ. of North Carolina, Chapel Hill, 1976.

[Howden 78]
W. E. Howden, Algebraic Program Testing, *Acta Informatica* **10**, 1978.

[Howden 80]
W. E. Howden, Functional Program Testing, *IEEE Trans. Software Engr.* **SE-6**, pp. 162-169, Mar. 1980.

[Howden 81]
W. E. Howden, A Survey of Dynamic Analysis Methods, pp. 209-231 in *Tutorial: Software Testing & Validation Techniques, 2nd Ed.,* ed. E. Miller and W. E. Howden, 1981.

[Hwang 81]

S-S. V. Hwang, An Empirical Study in Functional Testing, Structural Testing, and Code Reading/Inspection*, Dept. Com. Sci., Univ. of Maryland, College Park, Scholarly Paper 362, Dec. 1981.

[Johnson, Draper & Soloway 83]

W. L. Johnson, S. Draper, and E. Soloway, An Effective Bug Classification Scheme Must Take the Programmer into Account, *Proc. Workshop High-Level Debugging*, Palo Alto, CA, 1983.

[Linger, Mills & Witt 79]

R. C. Linger, H. D. Mills, and B. I. Witt, *Structured Programming: Theory and Practice,* Addison-Wesley, Reading, MA, 1979.

[McMullin & Gannon 80]

P. R. McMullin and J. D. Gannon, Evaluating a Data Abstraction Testing System Based on Formal Specifications, Dept. Com. Sci., Univ. of Maryland, College Park, Tech. Rep. TR-993, Dec. 1980.

[Mills 72]

H. D. Mills, Mathematical Foundations for Structural Programming, IBM Report FSL 72-6021, 1972.

[Myers 78]

G. J. Myers, A Controlled Experiment in Program Testing and Code Walkthroughs/Inspections, *Communications of the ACM,* pp. 760-768, Sept. 1978.

[Myers 79]

G. J. Myers, *The Art of Software Testing,* John Wiley & Sons, New York, 1979.

[Naur 69]

P. Naur, Programming by Action Clusters, *BIT* **9**, 3, pp. 250-258, 1969.

[Ostrand & Weyuker 83]

T. J. Ostrand and E. J. Weyuker, Collecting and Categorizing Software Error Data in an Industrial Environment, Dept. Com. Sci., Courant Inst. Math. Sci., New York Univ., NY, Tech. Rep. 47, August 1982 (Revised May 1983).

[Selby 84]

R. W. Selby, Jr., A Quantitative Approach for Evaluating Software Technologies, Dept. Com. Sci., Univ. Maryland, College Park, Ph. D. Dissertation, 1984.

[Selby, Basili & Baker 84]
R. W. Selby, Jr., V. R. Basili, and F. T. Baker, CLEANROOM Software Development: An Empirical Evaluation, Dept. Com. Sci., Univ. Maryland, College Park, Tech. Rep. TR-1415, July 1984. (submitted to the *IEEE Trans. Software Engr.*)

[Stucki 77]
L. G. Stucki, New Directions in Automated Tools for Improving Software Quality, in *Current Trends in Programming Methodology,* ed. R. T. Yeh, Prentice Hall, Englewood Cliffs, NJ, 1977.

THE VIEWGRAPH MATERIALS

for the

R. SELBY PRESENTATION FOLLOW

Evaluating Software Testing Strategies

Richard W. Selby, Jr. and Victor R. Basili
University of Maryland

Jerry Page
Computer Sciences Corporation

Frank McGarry
NASA/Goddard Space Flight Center

Overview

- **Problem:** The software community is uncertain of how to effectively test software

- **Idea:** Conduct a controlled study in which common testing techniques are applied to different types of software by a representative group of programming professionals.

- **Benefits:** Characterize how testing effectiveness relates to
  - different testing techniques
  - type of software being tested
  - type of faults in the software
  - interactions among testing techniques and type of fault or type of software

- **Action:** Organize and run controlled study (Oct. 1984)

Goals

Compare code reading, functional testing, and structural testing w. r. t.

- # faults detected
- cost-effectiveness
- classes of faults uncovered

Controlled Study

- Testing techniques: code reading, functional testing, and structural testing (stmt. cov.)

- Representative testing environment
  - 32 professional subjects from NASA/CSC (10 yrs.)
  - 3 programs (350, 170, 160 LOC)
  - faults (12, 9, 7)

- Iterative experimentation

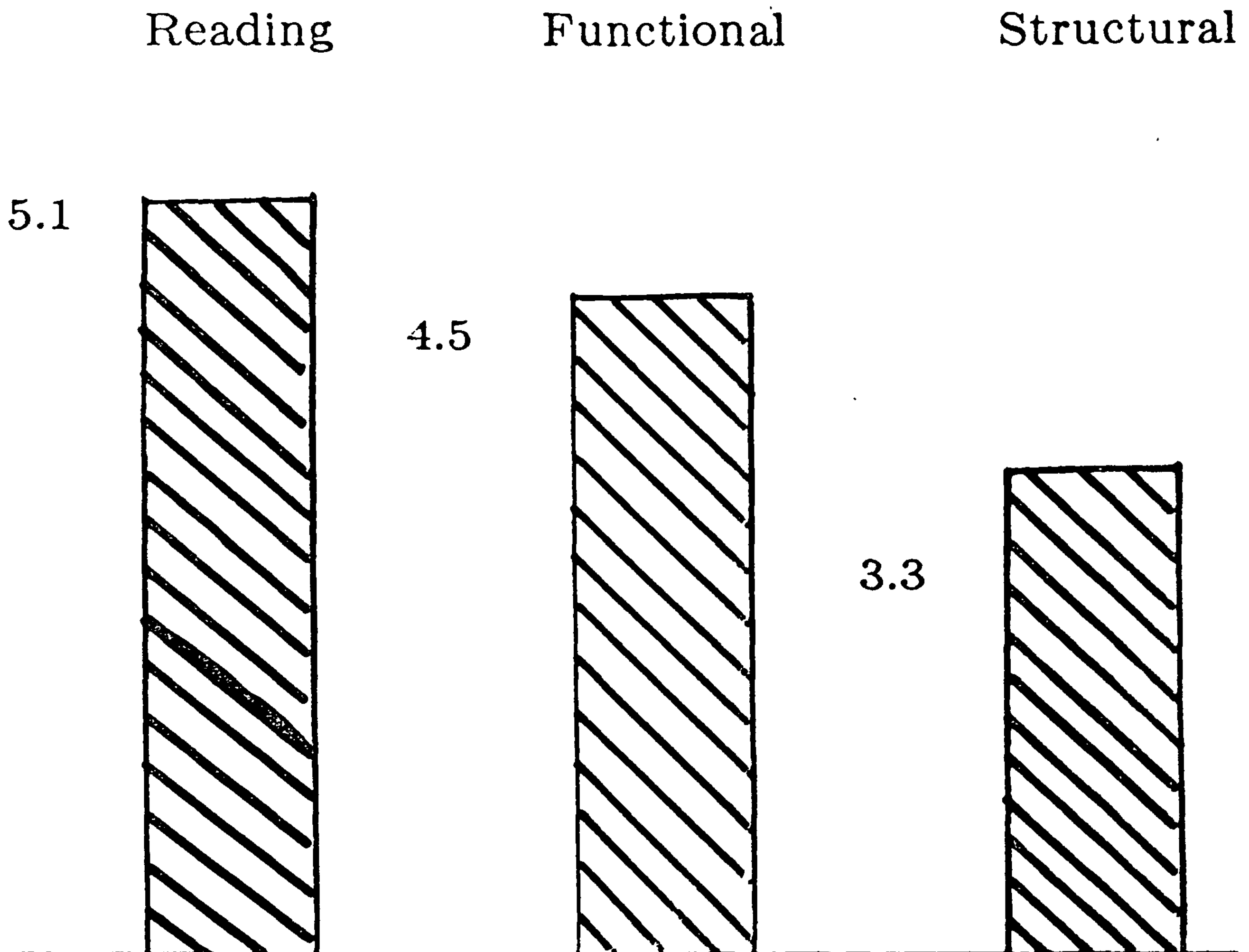
- Fractional factorial design

Fractional Factorial Design

		Code Reading	Functional Testing	Structural Testing
		P_1 P_2 P_3	P_1 P_2 P_3	P_1 P_2 P_3
Advanced Subjects	S_1	——X	—X—	X——
	S_2	—X—	X——	——X
	.		. . .	
	.		. . .	
	S_8	X——	——X	—X—
Inter-mediate Subjects	S_9	—X—	X——	——X
	S_{10}	——X	—X—	X——
	.		. . .	
	.		. . .	
	S_{19}	X——	——X	—X—
Junior Subjects	S_{20}	—X—	X——	——X
	S_{21}	X——	——X	—X—
	.		. . .	
	.		. . .	
	S_{32}	——X	—X—	X——

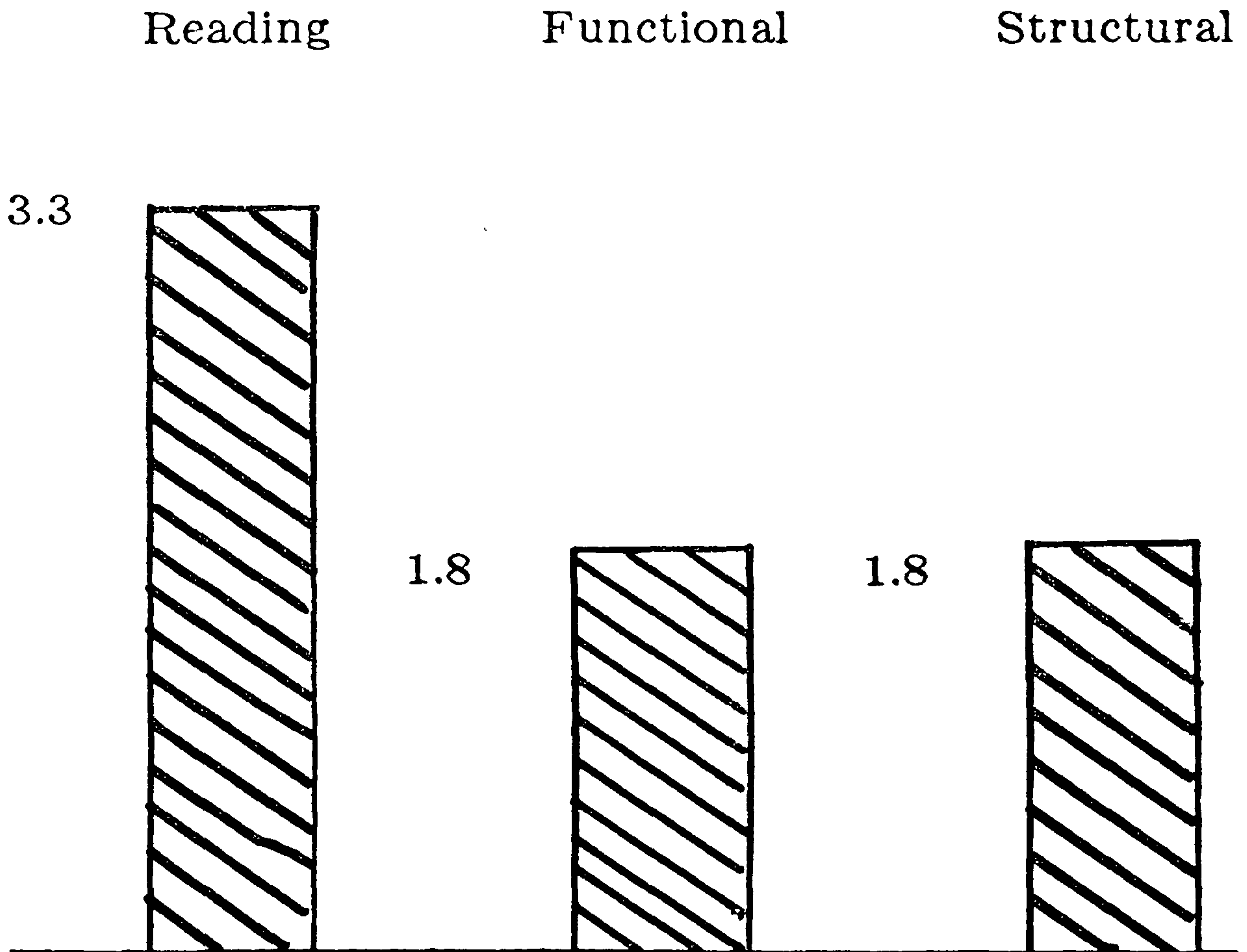
- Blocking according to experience level and program tested
- Each subject uses each technique and tests each program

Number of Faults Detected



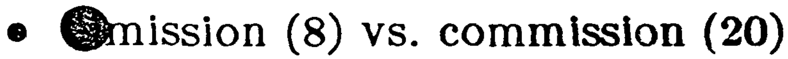
- Reading > others; Functional > Structural ($\alpha < .005$)
- Different # faults detected in each program
- Same relationships for % faults detected
- Advanced > others ($\alpha < .05$); Intermediate $\simeq$ Junior
- % detected correlates with % felt uncovered:
  $R = .57$ ($\alpha < .001$)

Cost-Effectiveness (#Faults Detected / Effort)



- Reading $>$ others ($\alpha < .005$, Est. $+1.5(.4)$);
  Functional $\simeq$ Structural
- Different overall detection rate for one program
- Techniques not different in total detection time
- Technique-program interaction ($\alpha < .005$)

Fault Characterization



- ●mission (8) vs. commission (20)

	Reading	Functional	Structural
100%	CCC●	CC●	
	CC	C	
	C	CC	C
	CC●C		C●
75%		●C	C
	C		
	CCCC	●CCCC	CCCC
		CC	C
50%	●		C
			CC
	C	C●	CCC
	C	CC	
	CC		
25%		C	CC
	●C	C	●C
	●		●
		●C●	●●
0%	●●C●	C●●C	C●●C●C

- Reading and functional stronger for omission faults

Fault Characterization

- Initialization (2-A), computation (4-P), control (6-C), data (2-D), interface (13-I), cosmetic (1-S)

	Reading	Functional	Structural
100%	IIIP	AIP	
	II	C	
	P	II	C
	AIIC		IP
75%		CP	I
	A		
	CPCC	CACPC	PIIC
		IC	A
50%	C		C
			PI
	I	II	AII
	S	II	
	IC		
25%		I	IC
	DI	S	CI
	I		D
		IID	CI
0%	PIID	IIDP	SDPIII

- Reading and functional stronger for control faults
- Reading stronger for interface faults

(Preliminary) Result Summary

	Code Reading	Functional Testing	Structural Testing
Detection Effectiveness	★★★	★★	★
Detection Rate	★★★	★★	★★
Total Detection Effort	—	—	—
Omission Faults	★★★	★★★	★★
Control Faults	★★★	★★★	★★
Interface Faults	★★★	★★	★★

Conclusions

- Each of the testing techniques showed merits in this representative evaluation

- Code reading performed well overall; functional testing similar in detection effectiveness

- Code reading learning curve (UMD studies)

- Related work
  - Hetzel
  - Myers
  - UMD studies
  - error studies
  - Cleanroom off-line development

- Use of these results

- Valuable insights into problems in software development and modification can be gained by controlled study

D3
N86-19970

SOFTWARE DEVELOPMENT IN ADA

Victor R. Basili
Elizabeth E. Katz
University of Maryland

1. Introduction

Ada will soon become a part of systems developed for the US Department of Defense. NASA must determine whether it will become part of its environment and particularly whether it will become a part of the Space Station development. However, there are several issues about Ada which should be considered before this decision is made. What information is needed to make that decision? What are the training needs for Ada? How should the life cycle be modified to use Ada most effectively? What other issues should management consider before making a decision? These are but a few of the issues that should be considered.

One means of considering these issues is the examination of other developments in Ada. Unfortunately, few full-scale developments have been completed or made publicly available for observation. Therefore, it will probably be necessary to study an Ada development in a NASA environment.

Another means related to the first is the development of Ada metrics which can be used to characterize and evaluate Ada developments. These metrics need not be confined to full-scale developments and could be used to evaluate on-going projects as well.

The remainder of this paper describes an early development in Ada, some observations from that development, metrics which have been developed for use with Ada, and future directions for research into the use of Ada in software development in general and in the NASA Goddard environment in particular.

2. Overview of a Previous Project

In a previous project conducted by the University of Maryland and General Electric, we monitored a software development project written in Ada by integrating measurement into the software development process. Our goal was to identify areas of success and difficulty in learning and using Ada as a design and coding language. The underlying process and the evolving product were measured, and the resulting information characterized this project's successes and failures. Observations from the project might be used to make recommendations about training, methodology, and metrics to the Ada users community. This experience with data collection and metrics also will aid in the selection of a general set of measures and measurement procedures for any software development project.

This work is supported in part by the Office of Naval Research and the Ada Joint Program Office under grant N00014-82-0225

Ada is a registered trademark of the US Department of Defense - AJPO

PRECEDING PAGE BLANK NOT FILMED

The project studied involved the redesign and reimplementation of a portion of a satellite ground control system originally written in FORTRAN. Four programmers were chosen for their diverse backgrounds and were given a month of training in Ada and software development methodology. They designed the project using an Ada-like PDL although a processor for the PDL was not available at that time. The design evolved into Ada code which was processed by the NYU Ada/Ed interpreter. The design and coding phases of the project extended from April 1982 to December 1982. Some unit testing of the project was done during the summer of 1983 using the ROLM compiler; however, the entire system has not been tested.

We used a goal-directed data collection approach from the beginning. Goals and objectives for the study were defined. Specific question and hypotheses were associated with each goal. Data collection forms and procedures were developed to address these questions. The forms and procedures were integrated into the software development methodology. The final step of this approach involved analyzing the data in order to answer the questions and either accept or reject the hypotheses.

Most recently, the data have been analyzed. All the data from the forms were entered in a database as were the data gathered by a processor which parses the design and code, checking for correct syntax and taking various measurements. Our observations are summarized below and elaborated upon in [2] and [3].

3. Observations from that Project

Although the project studied ended part way through development, the results indicate what might happen in early stages of development in other projects. The data can be compared with the corresponding stages of other projects. The results from this project may prevent others from making costly management mistakes.

Learning Ada takes time. In this project it consumed 20% of the total effort. That time must be included in any estimate of effort for early projects using Ada. Training will probably have to be a continuing process as the team members learn the finer points of the language.

Ada is more than syntax and simple examples. The underlying software engineering concepts must be taught in conjunction with the support Ada provides for those concepts. Most programmers are not familiar with the methodologies developed in the seventies that Ada supports. Training in software engineering methodology and how to use it in the environment of a particular application is an absolute necessity for the proper use of Ada.

We do not know how Ada should be used. Ideally, our understanding of the software engineering concepts Ada supports would make the use of Ada natural. However, many people learn by example, and we do not have many good examples of how Ada should be used. We do not know how and when to use exceptions, tasks, and generics. We need to study various alternatives and show how they work with examples from various environments.

Design alternatives must be investigated. The design for this project was functional and more like than unlike the earlier FORTRAN design. This may be the

best design, but a group at General Electric developed an object-oriented design for the same project [4]. It is not clear which design, if either, is most appropriate. Just as a combination of top-down and bottom-up development is appropriate to many applications, a combination of functional and object-oriented design might well be most appropriate. Only after we know which type of design, or combination thereof, is best suited to the particular application can we teach people which design approach to use. Without such training, programmers will rely on their experience with other languages and will probably produce functional designs.

Proper tool support is mandatory. This project was done without a production-quality validated compiler. In addition to that very necessary tool, a language-oriented editor, which could have eliminated 60% of the observed errors, would have been desirable. This would have allowed the programmers to focus their attention on the logic errors that undoubtedly remain in the design and code. Data dictionaries, call structure and compilation dependency tools, cross references, and other means of obtaining multiple views of the system would have helped. A PDL processor with interface checks, definition and use relation lists, and various metrics would also be helpful.

Some methodology must be followed for a project to be successful. The methodology and tools to be used should be understood before the project begins. The effect of the lack of good tools is mentioned above. In addition, the PDL was loosely defined until after design began. Effective design reading might have caught many of the errors. Even if we wanted to test this project after a compiler became available, we would have needed to create a test plan after the requirements were completed. However, that aspect of the methodology was deemed unimportant. The language is only one aspect of the environment and methodology. It cannot save a project in which the rest of the methodology is ignored.

We believe that this project is atypical in that it was done before a compiler was available and was not finished. However, it is typical in that training consumed an enormous amount of effort and the programmers were not familiar with the underlying software engineering concepts of Ada and that it might look like the beginning of many projects. The learning curve in methodology is quite large. As we study more projects that use Ada, we will learn how to teach it, how to use it, and where we might make mistakes. Until then, we need to study Ada and its use further.

4. Metrics for Ada

In conjunction with the project described above, a number of metrics specific to Ada have been developed. Some of these have been used to evaluate the use of packages on that project and the other design presented in [4]. Two of the package metrics characterize the visibility of packages and the use of data hiding via packages. These and other metrics for packages are further described in [5].

Other aspects of Ada might also be measured. Although we have not studied these in detail at this time, metrics for tasking might characterize the shared code and evaluate the use of concurrency. Metrics for exception handling might measure

the locality of the exception handlers or the complexity of those handlers. However, we must determine how these aspects of Ada should be used before we try to assign qualitative values to these measures.

In addition, we are developing a taxonomy of evaluative, predictive, and characteristic metrics that might be used for Ada projects in particular but also non-Ada software developments. Metrics are placed in eight categories which fall roughly into two groups. The first group contains the process categories such as resource use, changes, and environment. The second group contains the product categories such as size, control, data, language, and operation. This is but one example of a categorization, and determining which categories are most pertinent to one's environment is a difficult task. However, we attempt to provide a set of metrics which can be used in conjunction with the data collection paradigm described above.

In addition to the categorization, the taxonomy also contains a formalization for describing metrics via formula generators. This is a notation for describing sets of metrics so that the myriad of combinations of metrics can be discussed without enumerating them. An earlier version of this work appeared in [1], but a better formalization is being developed.

5. Future Work

Ada is a new language and it is only starting to be used. We do not know how to teach people to use Ada correctly. We do not even know how Ada should be used. However, we plan some further research into Ada in order to answer some of the questions above.

We plan to continue our work with Ada-specific metrics. We would like to apply these metrics to various projects and compare the measures to our perceptions of the projects. Also in this area, we would like to develop more elaborate metric tools.

Also in the area of tools, we plan to categorize tools and techniques by the faults they will prevent, the faults they will detect, the faults they might detect, and the faults they will not detect. If we know the types of faults code developed in this environment usually contains, we might be able to apply the appropriate tools or techniques to best discover those faults.

There were many drawbacks to the project presented above. The training should have contained specific and more detailed examples. A clearly defined methodology, incorporating Ada, should have been used. Finally, the project should have been taken to completion. We plan to monitor other large projects in which these problems have been corrected. At least one of these will probably be done in the NASA environment to determine how Ada fits into that environment.

In addition, we would like to study various design alternatives. Comparisons of when to use an object-oriented versus a functional design would probably help in Ada training. However, we currently do not know when each type of design should be used. We need to determine some means of comparing designs and evaluating the various alternatives. Controlled experiments would be one vehicle, along with

the larger projects, for these studies of design.

There many interesting problems associated with Ada. We are addressing only some of those problems. We welcome any comments on our research and encourage others to investigate these and other aspects of Ada.

6. Acknowledgements

We wish to thank John Bailey, Shih Chang, John Gannon, Elizabeth Kruesi, Nora Monina Panlilio-Yap, Connie Loggia Ramsey, Sylvia Sheppard, and Marvin Zelkowitz for their contributions as the other monitors of the GE project.

7. References

[1] Victor R. Basili and Elizabeth E. Katz, "Metrics of Interest in an Ada Development," IEEE Workshop on Software Engineering Technology Transfer, Miami, FL, April 1983, pp. 22-29.

[2] Victor R. Basili, Nora Monina Panlilio-Yap, Connie Loggia Ramsey, Shih Chang, and Elizabeth E. Katz, "A Quantitative Analysis of a Software Development in Ada," University of Maryland Computer Science Technical Report, UOM-1403, May 1984.

[3] Victor R. Basili, Elizabeth E. Katz, Nora Monina Panlilio-Yap, Connie Loggia Ramsey, and Shih Chang, "A Quantitative Characterization and Evaluation of a Software Development in Ada," submitted to *IEEE Computer.*

[4] A.G. Duncan, J.S. Hutchison, J.W. Bailey, T.M. Chapman, A. Fregly, E.E. Kruesi, D. Merrill, T. McDonald, and S.B. Sheppard, "Communications System Design Using Ada," Proc. 7th Intl. Conf. on Software Engineering, Orlando, FL, March 1984, pp. 398-407.

[5] John D. Gannon, Elizabeth E. Katz, and Victor R. Basili, "Metrics for Characterizing Ada Packages" under draft.

THE VIEWGRAPH MATERIALS

for the

V. BASILI PRESENTATION FOLLOW

SOFTWARE DEVELOPMENT IN ADA

Victor R. Basili
University of Maryland

with
Elizabeth Katz
Nora Monina Panlilio-Yap
Connie Loggia Ramsey
Shih Chang

and
other project members
John Bailey
John Gannon
Elizabeth Kruesi
Sylvia Sheppard
Marvin Zelkowitz

* This work is supported by the Office of Naval Research and the Ada Joint Program Office under grant N00014-82-0225.

+ Ada is a registered trademark of the United States Department of Defense - Ada Joint Program Office.

MOTIVATION

* Importance of studying Ada

  - NASA needs to make a decision about using Ada with the Space Station

  - How should people be trained?

  - How does background affect the learning and use of Ada?

  - How cost effective is the use of Ada?

* Look at other developments for suggestions

* Look at related projects for support in metrics and tools

UNIVERSITY OF MARYLAND / GENERAL ELECTRIC ADA PROJECT

* Redesign portion of satellite ground control
* Goal to make recommendations on training and tools
* Data collected according to paradigm
* Extensive training spread over a month
  - class, videotapes, practice project,
  - methodology then and during project
  - could have been more effective
* Used Ada-like PDL for design
* NYU Ada/Ed interpreter used for processing
* Project not completed and only partially tested due to lack of compiler

GOAL-QUESTION-METRIC PARADIGM FOR DATA COLLECTION

* Generate set of goals based on needs or organization
* Derive set of questions of interest or hypotheses which quantify those goals
* Develop a set of data metrics or distributions which provide the information needed to answer the questions
* Define a mechanism for collecting accurate data
* Validate the data
* Analyze the collected data to answer the questions

FOUR AREAS OF GOALS

* Characterize the effort, changes, errors, and Ada errors

* Evaluate the use of Ada, the effect of using an Ada-like PDL, the effect of programmer background on the use of Ada, and how much of Ada is used

* Evaluate the data collection and validation

* Develop a set of metrics for Ada and provide a data base for future Ada projects to predict some properties of those projects

PROGRAMMERS

Programmer	Years of Exper-ience	Education	Languages Known
Lead	9	B.S.	FORTRAN, Assembler
Senior	7	M.S.	FORTRAN, Assembler, SNOBOL, PL/1, LISP
Junior	0	B.S.	FORTRAN, Assembler, Pascal, PL/1, LISP
Librarian	0	High School	FORTRAN

* Had no experience with Ada
* Lead and senior programmer had some experience with the application but not with current software engineering techniques
* Junior programmer had most experience with newer software engineering techniques, and he created the made Ada-like code
* Each used the model of programming he knew best

PRODUCT CHARACTERISTICS

Ada and nonexpanded PDL	Programmer				
	Lead	Senior	Junior	Librarian	Total
nonblank lines	1633	3611	4307	396	9899
text lines	857	1904	2159	274	5154
executable stmts.	378	718	866	127	2089
compilation units	9	20	36	2	67

* Senior and junior programmers wrote most of the code

* Senior programmer expanded all of his design

* Librarian wrote very little code

* Some PDL was never expanded

* Design looked more like the original FORTRAN design than unlike it

MAKE THIS SIDEWAYS AT NASA

+ data analysis should look like it continues
+ half month is granularity
+ last row is hours (none for data analysis)

Date	requirements	training	design	code	test	data analysis	Activity
Jan 82	*						
Feb							
		*					
March	*	*					
	*	*					
April	*			*			
	*			*			
May			*	*	*		
			*				
June			*				
			*				
July			*				
			*	*			
Aug			*	*			
			*	*			
Sept			*	*			
				*			Calendar Time
Oct				*			
				*			
Nov				*			
				*			
Dec				*			
				*			
Jan 83						*	
						*	
Feb						*	
						*	
March						*	
						*	
April						*	
						*	
May					*	*	
					*	*	
June					*	*	
					*	*	
July					*	*	
					*	*	
Aug					*	*	
						*	
	684	849	714	381	332		Hours

CHANGES AND ERRORS

* Study changes and errors to determine problem areas and effectiveness of training

* Difficult to compare with completed projects

* 332 changes -- 57% were error corrections

* 192 errors

* Most errors were syntax errors and trivial

    - 90% affected only one component

    - 80% were isolated and corrected in less than half an hour

* .091 errors per executable statement

* Reading focussed on syntax errors rather than on more serious ones

OMISSION VS. COMMISSION ERRORS

* Compare with Basili & Perricone SEL study

Errors Involved	Percentage	
	Omission	Commission
This study w/o compiler faults	52%	48%
Basili & Perricone New module errors	45%	55%

BUT

* All Ada modules were new
* All SEL modules had at least clean compiles before error reporting began
* We don't consider those faults that could have been detected by a compiler

LANGUAGE-PROBLEM-CLERICAL ERRORS

* Subjective in that the monitors must try to determine what the programmer was thinking

* LANGUAGE - related to the use of Ada

  - SYNTAX - misunderstanding or misuse of the syntax of a feature
  - SEMANTICS - misunderstanding of the meaning of a feature in that language
  - CONCEPT - involves the general idea of how the feature should be used

* PROBLEM - misunderstanding of the problem domain or the environment

* CLERICAL - due to carelessness, e.g. typos

Category	Number of Errors	
	All Errors	w/o Compiler Faults
Language	160	18
Concept	8	8
Semantics	44	10
Syntax	108	0
Problem	26	26
Clerical	6	0
Total	192	62

* Language errors are rare for NASA projects

USE OF ADA FEATURES

* Most features were used, but not together
  - Generics were instantiated once
  - Some simple exception handling
  - Several tasks
  - No new abstract data types defined
* Little information hiding
  - Little private data
  - Representation of structures was shared
  - Changes to representation would be disastrous in some cases
* No attempt to limit visibility of data
* Packages for device drivers
* Ada-specific features were more error-prone

RECOMMENDATIONS FROM THIS STUDY

* Ada is more than syntax and simple examples

    - Learning Ada takes time

    - Need examples from application area

    - Ada should be used with some methodology

    - Need training in methodology

* Lessons in tool support

    - Must evaluate quality and availability of compilers and other tools

    - Language-oriented editor would alleviate the problems with syntax errors

* Design alternatives should be investigated

* Study how Ada features should be used

METRIC DEVELOPMENT

* Look for metrics to evaluate methodology

* Metrics to evaluate Ada use

  - Package metrics (Gannon, Katz, and Basili)

    + Visibility
    + Implementation hiding

  - Tasking metrics

    + Shared code
    + Concurrency

  - Exception metrics

    + Locality of handlers
    + Complexity of handlers

* General metrics are available for evaluating other aspects of the development

* Metric taxonomy of evaluative, predictive, and characteristic metrics (Basili and Katz)

  - Eight categories in two groups

PROCESS	PRODUCT
resource use	size
changes	control
environment	data
	language
	operation

  - Formalization via formula generators

C-2

FUTURE WORK

* Continue work on Ada-specific metrics
* Develop more elaborate metric tools
* Categorize tools and techniques by the faults they prevent, will detect, might detect, or will not detect
* Monitor other large projects, e.g. NASA
    - Training with specific examples
    - Clearly defined methodology
    - Taken to completion
* Study design alternatives
    - When to use object-oriented vs. functional design
    - How to evaluate alternatives
    - Controlled experiments

PANEL #2

SOFTWARE ERROR STUDIES

J. Knight, University of Virginia
H. Rombach, University of Maryland
R. Sum, University of Illinois
E. Soloway, Yale University

D4

N86-19971

A LARGE SCALE EXPERIMENT IN N-VERSION PROGRAMMING

John C. Knight
Department of Computer Science
University of Virginia
Charlottesville, Virginia.

Nancy Leveson
Department of Computer Science
University of California
Irvine, California.

A Summary

Submitted To The Ninth Annual Software Engineering Workshop
Goddard Space Flight Center
Greenbelt, Maryland.

N-version programming has been proposed as a method of providing fault tolerance in software. The approach requires the independent preparation of several (i.e. "N") versions of a piece of software for some application from the same requirements specifications. These versions are executed in parallel in the application environment; each receives identical inputs and each produces its version of the required outputs. The outputs are collected by a voter and, in principle, they should all be the same. In practice there may be some disagreement. If this occurs, the results of the majority are assumed to be the correct output and this is the one used by the system.

The great benefit that N-version programming is intended to provide is a substantial improvement in reliability. It is assumed in the analysis of the technique that the N different versions will fail *independently*; that is faults in the different versions occur at random and are unrelated. Thus the probability of two or more versions failing on the same input is very small. Under this assumption, the probability of failure of an N-version system, to a first approximation, is proportional to the N'th power of the probability of failure of the independent versions. If the assumption is true, systems with extremely high levels of reliabilty could be built with components that are individually of only average quality.

We are concerned that this assumption might be *false*. Our intuition indicates that when solving a difficult intellectual problem (such as writing a computer program), people tend to make the same mistakes even when they are working independently. If the assumption of independence is not born out in practice, it would cause the analysis of the reliability to overestimate the reliability of an N-version system. This could be an important practical problem since N-version programming is being used in existing crucial systems and is planned for others.

To test this underlying assumption of independence, we have carried out a large scale experiment in N-version programming. A *statistically rigorous* test of independence was

the major goal of the experiment and all of the design decisions that were taken were dominated by this goal.

In graduate classes in computer science at the University of Virginia (UVA) and the University of California at Irvine (UCI), students were asked to write programs from a single requirements specification. The result was a total of twenty seven programs (nine from UVA and eighteen from UCI) all of which should have produced the same output from the same input. Each of these programs was then subjected to one million randomly generated test cases.

The problem that was selected for programming is a simple simulation of an anti-missile system. The program is required to read some data that is supposed to represent radar reflections. Using a collection of conditions, the program has to decide whether the radar reflections come from an object that is a threat or otherwise. If the decision is made that the object is a threat, a signal to launch an interceptor has to be generated. The problem is known as the "launch interceptor" problem and the various conditions upon which the decision depends are referred to as "launch interceptor conditions" (LIC's). The various conditions are heavily parameterized. For example, one condition asks whether a set of reflections can be contained within a circle of given radius; the radius is a parameter.

The students were given a brief explanation of the goals of the experiment and the principles of N-version programming. The need for independent development was stressed and students were carefully instructed not to discuss the project amongst themselves. However, we did not impose any restriction on their reference sources. Since the application requires some knowledge of geometry, it is to be expected that the students would consult reference texts and perhaps mathematicians in order to develop the necessary algorithms. We felt that the possibility of two students using the same reference material was no different from two separate organizations using the same reference sources in a commercial development environment.

As would be expected during development, questions arose about the meaning of the requirements (surprisingly few questions we are pleased to say). In order the prevent any possibility of information being inadvertently transmitted by an informal verbal response, these questions were submitted and answered by electronic mail. If a question revealed a general flaw in the specifications, the response was broadcast to all the programmers.

Each student was supplied with twelve input data sets and the expected outputs for use in debugging. Once a program was debugged using these tests and any other tests the student developed, it was subjected to an acceptance test. The acceptance test was a set of two hundred randomly-generated test cases: a different set of two hundred tests were generated for each program. This procedure was used to prevent a general "filtering" of common faults by the use of a common acceptance test. Once a program passed its acceptance test, it was considered complete and entered into the collection of versions.

Once all the versions had passed their acceptance tests, program development was stopped and the versions were tested. A test driver was built which generated random radar reflections and random values for all the parameters in the problem. All twenty seven programs were executed on these test cases and the determination of success was made by comparing their output with a twenty-eighth version, referred to as the *gold* program. The gold program had been tested extensively in other experiments and had been the subject of an extensive walkthrough. It was thought to be correct but each disagreement between the gold version and one of the others was investigated to ensure that the gold version was not at fault. A total of one million tests were run on these twenty eight versions.

For the particular problem that was programmed for this experiment, we have concluded, based on the results on the million tests, that the assumption of independence that is fundamental to the analysis of N-version programming *does not hold*. Using a fairly simple probability model of independence, our data indicates that the hypothesis of

independence has to be rejected at the 99% confidence level.

It is important to understand the meaning of this statement. First, it is conditional *on the application that we used*. The result may or may not extend to other programs, we do not know. Other experiments must be carried out to gather data similar to ours in order to be able to draw *general* conclusions. Second, the statement above does not mean that N-version programming does not work or should not be used. It means that the reliability of an N-version system *may* not be as high as theory predicts under the assumption of independence. If the implementation issues can be resolved for a particular N-version system, the required reliability might be achieved by using a larger value for N.

THE VIEWGRAPH MATERIALS

for the

J. KNIGHT PRESENTATION FOLLOW

A LARGE-SCALE EXPERIMENT IN N-VERSION PROGRAMMING

John C. Knight
Department of Computer Science
University of Virginia
Charlottesville, Virginia, 22903
(804) 924-7605

Nancy G. Leveson
Department of Computer Science
University of California
Irvine, California, 92717
(714) 856-5517

LARGE-SCALE EXPERIMENT IN N-VERSION PROGRAMMING

- Fault-Tolerant Software By N-Version Programming
- Currently Being Applied (A310 AIRBUS)
- Examination of Assumption of Independence
- Statistically Rigorous Analysis
- Two Universities - UVA and UCI
- Graduate and Senior Classes in Software Engineering Provided Programmers

EXPERIMENT OVERVIEW

- Specifications Rewritten at UVA Based on RTI Experience
- RTI Gold Program Rewritten in Pascal
- Twenty Seven Versions Written
- Each Required To Pass 200 Test Cases Before Acceptance
- Satisfactory Versions Subjected to One Million Tests
- Intermediate Computations Checked
- 7 VAX's, 5 Primes, 2 Cyber 170's, Cyber 730

PROGRAM OVERVIEW

- Processing of Simulated Radar Data
- Considerable Geometric Knowledge Required
- Written in Pascal
- Final Versions Turned Out To Be 500 - 800 Lines
- Versions Written as Procedures
- All I/O Through the Parameters
- Fixed Precision Real-Compare Function

VERSION FAILURE DATA

Version	Failures	Reliability	Version	Failures	Reliability
1	2	0.999998	15	0	1.000000
2	0	1.000000	16	62	0.999938
3	2297	0.997703	17	269	0.999731
4	0	1.000000	18	115	0.999885
5	0	1.000000	19	264	0.999736
6	1149	0.998851	20	936	0.999064
7	71	0.999929	21	92	0.999908
8	323	0.999677	22	9656	0.990344
9	53	0.999947	23	80	0.999920
10	0	1.000000	24	260	0.999740
11	554	0.999446	25	97	0.999903
12	427	0.999573	26	883	0.999117
13	4	0.999996	27	0	1.000000
14	1368	0.998632			

MULTIPLE FAILURES

Number	Probability	Occurrences
2	0.00055100	551
3	0.00034300	343
4	0.00024200	242
5	0.00007300	73
6	0.00003200	32
7	0.00001200	12
8	0.00000200	2

CORRELATED FAILURES - UCI AND UVA

		UVA Versions								
		1	2	3	4	5	6	7	8	9
UCI Versions	10	0	0	0	0	0	0	0	0	0
	11	0	0	58	0	0	2	1	58	0
	12	0	0	1	0	0	0	71	1	0
	13	0	0	0	0	0	0	0	0	0
	14	0	0	28	0	0	3	71	26	0
	15	0	0	0	0	0	0	0	0	0
	16	0	0	0	0	0	1	0	0	0
	17	2	0	95	0	0	0	1	29	0
	18	0	0	2	0	0	1	0	0	0
	19	0	0	1	0	0	0	0	1	0
	20	0	0	325	0	0	3	2	323	0
	21	0	0	0	0	0	0	0	0	0
	22	0	0	52	0	0	15	0	36	2
	23	0	0	72	0	0	0	0	71	0
	24	0	0	0	0	0	0	0	0	0
	25	0	0	94	0	0	0	1	94	0
	26	0	0	115	0	0	5	0	110	0
	27	0	0	0	0	0	0	0	0	0

FAULTS DETECTED DURING TESTING

Version	Faults	Version	Faults
1	1	15	0
2	0	16	2
3	4	17	2
4	0	18	2
5	0	19	1
6	3	20	2
7	3	21	2
8	2	22	3
9	2	23	2
10	0	24	1
11	1	25	3
12	2	26	8
13	1	27	0
14	2		

- Bug (a) Shared By 1, 18(twice)
- Bug (b) Shared By 3(twice), 8(twice), 20, 25(twice)
- Bug (c) Shared By 7, 12, 14, 17(twice)
- Bug (d) Shared By 9, 11,20, 22(3 times), 26(twice)
- Bug (e) Shared By 13, 16, 21

DISCUSSION

- Student Programmers Are Realistic Subjects
- Million Tests Represent Reasonable Lifetime
- Conclusions
  - Computed Probability of Multiple Failures - 0.000126
  - Observed Probability of Multiple Failures - 0.001255
  - Hypothesis of Independence Rejected at the 99% Level
  - N-Version Programming Needs to be Used CAREFULLY
  - Many More Experiments Are Needed

N86-19972
D5

DESIGN METRICS FOR MAINTENANCE +

H. Dieter Rombach *

Department of Computer Science
University of Maryland
College Park MD 20742
(301) 454-4251

Abstract

This paper describes results of a study to develop **maintenance metrics** based on **structural software design characteristics**. The intent of the study was to define a **characteristic metric set**, suited to **explain** and **predict** software maintenance behavior. The maintenance aspects investigated in this study are **stability** and **modifiability**. While stability addresses the average number of modules affected per change cause, modifiability characterizes the ease with which changes can be made within each of these modules. Additional interest is dedicated to the difference between characteristic **design** and **implementation** metric sets, and to the difference between change behavior during **development** and **maintenance**. This study examines the development of six software systems and controlled maintenance experiments using these systems.

---

* Some of these results are contained in my Ph.D. thesis [Rombach 84] written at the Dept. of Computer Science, University of Kaiserslautern, Fed. Rep. of Germany.

+ Research for this study was supported in part by the ministry of research and technology of the Fed. Rep. of Germany (Project on DISTributed Operating Systems at the University of Kaiserslautern, Fed. Rep. of Germany).

Motivation

The study presented in this paper was part of a project to design and implement a new LAnguage for DIstributed sYstems (LADY [Nehmer et al. 82]), started at the Computer Science Department at the University of Kaiserslautern, Federal Republic of Germany, in 1980.

The overall goals of this project were to improve the behavior of software for distributed systems with respect to comprehensibility, maintainability, and reusability. To achieve these goals, the following language features were included:

1) A hierachy of two explicit levels to structure a system: A system is characterized as a set of teams (functional units of distribution), each team as a set of modules (units of separate compilation).
2) Strong typing, even of structural units.
3) Formal interface parameters.

To determine the degree to which these goals were met, quantitative estimation of the behavior of systems was developed. The behavior of a number of systems implemented in LADY were compared with a number of systems implemented in a 'conventional' language without these features. The behavior of software is influenced by various factors [Basili 81]. In order to attribute different behavior to system structure, it was necessary to keep all factors not of interest as constant as possible. One way to achieve this is to use restrictive development and documentation guidelines, if possible supported by tools. One of the tools [Rombach, Wegener 84] was used to assist in developing consistent, semiformal design documents based on ideas in [DeRemer, Kron 76]. On the other side, this increase of formalism of design documents was the presupposition to extend research about the influence of measurable structural software characteristics on software behavior from code to design documents.

This paper focuses on the quality characteristic maintainability and its predictability by structural design characteristics. Data for this study were collected from six systems, all designed and implemented using the above mentioned 'conventional' modularization concept.

Goals

The overall goal of this study is to determine the impact of structural software design characteristics on maintenance behavior. Before stating the questions of interest, a few terms have to be introduced: Each failure, change of environment, or change of requirements is called a **change cause.** Each change cause can result in a number of **changes** in different modules. Analogously, two different maintenance aspects, **stability** and **modifiability**, are of interest. While stability addresses the impact of each change cause on the whole system, e.g., number of affected modules, modifiability characterizes the ease with which changes can be made within each of these single modules. For each module, the effort spent to change this unit is called its **internal change effort.** The effort spent in all other units because of the same change cause is called its **external change effort.**
The **questions of interest** are:

(Q_1) Is it possible to explain or predict stability in terms of 'number of changed units per change cause during maintenance' by analysis of the system structure as available from design documents?

(Q_2) Is it possible to explain or predict stability in terms of 'external change effort in staff_hours per change cause during maintenance' by analysis of the system structure as available from design documents?

(Q_3) Is it possible to explain or predict modifiability in terms of 'inter-

nal change effort in staff_hours per change during maintenance' by analysis of the system structure as available from design documents?

Two additional questions address the impact of the terms 'design document' and 'maintenance' in the three questions above:

(Q_4) Are questions (Q_1) to (Q_3) answered differently, if software characteristic data to characterize system structure are collected from code instead of design documents?

(Q_5) Are questions (Q_1) to (Q_3) answered differently, if changes are analyzed during development instead of maintenance?

Software Model

Very different models as abstractions of software depending on the aspect of interest [Harrison 82], [Henry, Kafura 81]. An **information_flow based model** seems to be most sensitive regarding all inter_module aspects, as specified by the type of design document used in this study (see chapter 'Experimental Approach'). Based on a model presented in [Henry, Kafura 81], a software system is modelled as a set of algorithmic units (modules) and global data, and various information flows between these modules.

An information flow from module A to module B is of type

1) **Explicit Global,** if "A has write_access and B has read_access to the same global variable".
2) **Implicit Global,** if "B uses information from module A, not explicitly available as data in code".

   This implicit global information flow is added to the original model because it seems to be a very important aspect, especially (but not only) in distributed

systems. Examples of such flows are shared assumptions about environment parameters such as number of terminals or assumptions about buffer sizes. In most cases, these implicit dependencies are the result of design decisions not specified at all or lost by transforming designs to code. This undocumented information can be expected to cause problems, if personnel not involved in the development of a system have to change this system.

3) **Local Direct**, if "A calls or uses B".

4) **Local Indirect**, if either a) "B receives data from A, caused by a call_ or use_relation from B to A", or b) "A is connected by local indirect flow of type a) to a third module C, and C calls or uses B with the same data received from A".

In this study, one of the important aspects with regard to the practical usability of metrics is, whether possible metrics are determined by automatically measurable data, or whether additional data are needed, which have to be analyzed or even determined by intuition.

Regarding this, a grouping of the above mentioned flows in 3), 1) + 4), and 2) seems to be adequate. Further in this study, the following significant terms will be used to classify metrics as based on:

- **Control Flow**, if only flows of type 3) are considered.
- **Data Flow**, if flows of type 1) and/or 4) are needed in addition to flows of type 3).
- **Information Flow**, if flows of type 2) are needed in addition to flows of type 1), 3), 4).

In the given order, the number of structural aspects taken in consideration increases, and the ease of collecting the necessary data decreases.

Experimental Approach

3 Time Sharing Systems (TSS), all implementing identical requirements, and 3 Process Control Systems (PCS), all implementing identical requirements, were developed and maintained to collect data in order to answer the questions (Q_1) to (Q_5).

- **Experimental Design**

The experiments were carried out in 2 subsequent steps.

Step_1, the development of these systems, was done by three graduate students (assisted by a number of student research assistents) writing their diploma (master) theses. These developments took about 18 months. The developed systems are characterized in Table 1.

Step_2, a number of controlled maintenance experiments, (as) identical (as possible) for all six systems developed in step_1, was done by student research assistents over about 6 months. First, the systems were seeded with **25 faults** of different types which the students had to isolate and correct.

The selection criterion for all faults was to choose a distribution pattern of fault types (control flow, data flow, data structure, computation, etc.) corresponding to the average one determined for all systems during development. All the faults to be isolated and corrected were specified by a system specific failure description. Second, the students had to adapt the systems to **10 changes of environment**, e.g., new interface to devices. Third, the students had to carry out **15 changes of system requirements**. None of the students involved in step_2 of the experiments for a specific system was involved in step_1 for this specific system. So, maintenance experiments were carried out for each system by students getting all their knowledge about the systems exclusively from existing documents.

- **Experimental Environment**

  The **design language** used, forces all decisions to be described explicitly. It is based on a module_interconnection_language presented in [DeRemer, Kron 76]. That means not only one final design version is described, but a number (depending of the developer's capability to handle the problem) of (different abstract) design views are described. Therefore, the whole design documentation consists of a hierarchy of, single_level_descriptions (see Figure 1). Figure 2 outlines the scheme of the complete module design document. Such a module design description (= description of level_n in Figure 1) contains formal specification of the module interface (EXPORT, IMPORT in Figure 2) and the algorithmic design (DYNAMICS in Figure 2) at least. Descriptions of different levels in Figure 1 usually consist of a different portion of formal information. Descriptions of level_1 to _n-1 differ in the sense from level_n descriptions, in that the complete implementation part doesn't exist yet. The specification part already exists, perhaps in a more abstract view depending on the level of the design document within the hierarchy of Figure 1.

  The **implementation language** used is an extension of PASCAL (plus concurrent processes and communication primitives), called C-TIP, which consists of 2 structuring levels:

  - system level (specification), describing a system as a set of modules (processes, classes, procedures), processes only communicating by exchanging messages
  - module level (algorithmic), like PASCAL (plus communication primitives)

- **Data Collection**

  Data were collected both to characterize the software structure and to characterize the software maintenance behavior.

  A list of Structural software design Characteristics (SC_i), for which data were

collected per module, is:

SC_1) Number of exported functions

SC_2) Number of parameters per exported function

SC_3) Number of imported functions

SC_4) Number of parameters per imported function

SC_5) Number of exported functions with output parameters

SC_6) Number of imported functions with output parameters

SC_7) Number of exported implicit informations (= flow of type 2))

SC_8) Number of imported implicit informations (= flow of type 2))

SC_9) Number of other modules 'using' exported functions

SC_10) Number of other modules implementing the imported functions

SC_11) Number of other modules 'using' exported functions with output parameters

SC_12) Number of other modules implementing imported functions with output parameters

SC_13) Number of other modules establishing implicit information to be used

SC_14) Number of other modules to which the observed module has information flow relations

These structural software design characteristics data were collected after development (characterizing the structure of the final version of a system).

A list of Quality Characteristics (QC_i), characterizing the maintenance behavior, is:

QC_1) Number of modules changed per change cause

QC_2) Effort in staff_hours to isolate per change cause

QC_3) Effort in staff_hours to correct or change in each module per change cause

These quality characteristic data were collected during all phases of development (starting with design) and during maintenance experiments with a separate form for each change cause.

- **Data Validation**

Validation of collected data was carried out by the author meeting with all developers at the beginning of each week.

- **Data Evaluation**

Although the study concentrates on the inter module aspect of system structure, the metrics under investigation combine this **exterior** complexity (coupling [Myers 75], programming_in_the_large [DeRemer, Kron 76]) with the **interior** complexity (strength or cohesion [Myers 75], programming_in_the_small [DeRemer, Kron 76]).
Therefore, for each module these complexity metrics K are of type

$$K \sim K_{exterior} * K_{interior}$$

The **exterior complexity** is composed of two views:

- **integrated view**, that considers the module embedded in an concrete system. Actual flows between this specific module and its environment are considered. Software characteristics 9) - 14) especially contribute to this aspect. Depending on which type of flow is of interest, the exterior complexity is represented by one of the following combinations of structural software design characteristics: "SC_9 + SC_10", "SC_11 + SC_12", "SC_13 + SC_14", etc..
- **isolated view**, that considers the module isolated (library module for future and different use). Possible flows between this specific module and its environment are considered. Software characteristics 1) - 8) especially contribute to this aspect. Depending on which type of flow is of interest, the

exterior complexity is represented by one of the following combinations of structural software design characteristics: "(SC_1 * SC_2) + (SC_3 * SC_4)", "SC_1 + SC_3", etc..

The **interior complexity** is composed of three measures:

- **Structure "v (G)"** (like Cyclomatic Complexity [McCabe 76])
- **Design Length "L"** in terms of the number of linear internal program sequences.

  A program is represented by a graph as in [McCabe 76], except that nodes are not only nonlinear control constructs, but also interface accesses (export_, import_functions). L is the number of edges of the corresponding graph.
- **Number of Interface Accesses "IA"** in terms of number of calls of import functions (see Figure 2) plus number of exported functions.

In order to answer the questions of interest (Q_1), data were evaluated in the following way:

- Determine for all modules of each system the **correlations** between the module complexity and the module-specific quality characteristic data (**QC_i**) for all faults during development.
- Determine for all modules of each system the **correlations** between the module complexity and the module-specific quality characteristic data (**QC_i**) for all maintenance experiments, including faults, environment adaptions, requirement changes (dangerous, because number and type of changes were fixed by intuition!)

The correlation between structural design characteristics and quantitative quality characteristics is determined by using the **Spearman correlation coefficient** together with its **level of significance**.

Data Analyses Results

All analysis results are presented according to the questions of interest:

- **Answers to question (Q_1):**

For all modules of each system, the correlations between different types of 'module complexity' and the 'average number of modules changed because of all change causes, the corresponding module was changed too' are presented.
All the results are supported by the data in Table 2, row 3 and 4, for two representative systems.

- The overall correlations are sufficiently good for analyzing not the completely implemented system but only design documents. The best correlation coefficients for each type of metrics are between 0.7128 and 0.8200 (significance < 0.01 at least).
- The best metrics to explain this stability aspect are using 'integrated information flow' to characterize the exterior complexity and the 'number of interface accesses IA' to characterize the interior complexity. The very best correlation coefficient is 0.8200 with significance level 0.001.
- The best metric using the 'integrated data flow' is not significantly worse than the best metric based on 'integrated information flow'.
- Metrics using the 'isolated data or information flow' show worse correlations than metrics using 'integrated data or information flow'.
- Metrics not using any characterization of the interior complexity are in the range between 0.5494 and 0.7180 (significance in most cases 0.05 at least).
- Conventional metrics, using the interior complexity such as 'v(G)' and 'L', show no sufficient correlation. Only 'IA', the characterization of the intensity of interface access, has a sufficiently good correlation with number of changes. This fact is reflected in the fact that all metrics which characterize the interior complexity by 'IA' result in the highest correlation coefficients.

- **Answers to question (Q_2):**

For all modules of each system, the correlations between different types of 'module complexity' and the 'average external change effort in staff_hours per change cause' are presented.

All the results are supported by the data in Table 2, row 5 and 6, for two representative systems.

The results overall are comparable to those corresponding to question (Q_1). The same pattern can be recognized, which says that for each exterior complexity class the metric using the 'number of interface accesses IA' shows the best correlation.

- The overall correlations were sufficiently good for analyzing not the completely implemented system but only design documents. The best correlation coefficients for each type of metrics are between 0.6643 and .8065 (significance < 0.05 at least).
- The best metrics to explain this stability aspect are using only the 'integrated information flow' to characterize the exterior complexity. The very best correlation coefficient is 0.8065 with significance level 0.001.
- The best metric using the 'integrated data flow' is not much worse than the best metric based on 'integrated information flow' (0.7780).
- Metrics using the 'isolated data or information flow' show worse correlations than metrics using 'integrated data or information flow'.
- Conventional metrics, using the interior complexity such as 'v(G)' and 'L', show no correlation. Only 'IA', the characterization of the intensity of interface access, has a sufficient correlation with number of changes. This fact is reflected in the fact that all metrics which characterize the interior complexity by 'IA' result in a higher correlation coefficient than those using 'v (G)' or 'L'.

- **Answers to question (Q_3):**

  For all modules of each system, the correlations between different types of 'module complexity' and the 'average internal change effort in these modules per change cause' are presented.

  All the results are supported by the data in Table 3 for four representative systems.

  - The overall correlations were sufficiently good for analyzing not the completely implemented system but only design documents. The best correlation coefficients for each type of metrics are between 0.6901 and 0.8230 (significance < 0.05 at least).
  - The best metrics to explain this stability aspect are using the 'integrated information flow' to characterize the exterior complexity and the 'length L' to characterize the interior complexity. The very best correlation coefficient is 0.8230 with significance level 0.001.
  - The best metrics using the 'integrated data flow' still show sufficiently good correlations (0.6984 to 0.7962).
  - Metrics using the 'isolated data or information flow' show no worse correlations than metrics using 'integrated data or information flow'.
  - Metrics not using any characterization of the interior complexity are in the range between 0.6901 and 0.7870 (significance in most cases 0.05 at least).
  - Conventional metrics, using the interior complexity such as 'v(G)' and especially 'L' show sufficiently good correlation. 'IA', the characterization of the intensity of interface access, doesn't correlate with number of changes at all. This fact is reflected in the fact that all metrics which characterize the interior complexity by 'L' result in the highest correlation coefficients.

All results corresponding to questions (Q_1) to (Q_3) are supported by results about change behavior of the systems during development. These results are

presented in detail in [Rombach 84]. The following data analysis results corresponding to questions (Q_4) and (Q_5) are **not** supported by data presented in this paper but by data in [Rombach 84]. Nevertheless, the results are presented briefly because it might help to put the results about design metrics for maintenance in perspective.

- **Answers to question (Q_4):**

  The same correlation pattern exists for metrics using structural data from code documents as for those using structural data from design documents.

  - The correlation coefficients using data from code are about 0.1 higher.

    It always must be remembered that the reported good results for design metrics depend very much on the formal way of documenting designs used in this study.

- **Answers to question (Q_5):**

  Correlation coefficients show a simillar pattern for all change causes during development as for maintenance experiments.

  - The total change effort for maintenance experiments was about twice as high as for the same changes during development.
  - The ratio 'isolation effort/change effort' was about 1:1 during development and about 3:1 for maintenance experiments.

Use of Analyses Results

Fortunately, **design metrics** characterized only by **explicitly measurable or analyzable** structure data show sufficiently high correlation with **stability** and **modifiability**. The best complexity metrics of this type explain **stability** of a module by its **data flows with other modules** (integrated data flow), and the **number of internal interface accesses ('calls') appearing in its module design. Modifiability** is explained by the same **integrated data flow** as

stability and by the **design length** of the algorithmic module design. These metrics can be completely **automated** and used at the **end of module design** as a

- **development tool,** to decide between design alternatives in a ordinal way, or
- **management or quality assurance tool,** to plan module specific testing effort according to complexity, or to consider redesign if module complexity exceeds tolerable complexity bounds.

The fact that complexity metrics only characterized by **exterior** aspects like **integrated data flow** still show sufficiently good correlations encourages the use of these design metrics for maintenance not only at the end of module design but much earlier during design. They should be used as soon as a system design, which describes the module interactions in some formal way, exists.

The main result of this study can be summarized as follows: **Software structure proved to be a reliable base to explain maintenance behavior. The result can be improved if the amount of implicit information in design documents can be decreased by forcing the principle of explicit documentation of all design decisions.** This is true because the conversion of implicit global data to explicit global data makes this information explicitely measurable so that no difference exists between **data flow** metrics and **information flow** metrics in Table 2 and 3. This result has been validated by controlled experiments under the described experimental environment. Especially, the necessary requirements for formal design documents have to be reminded.

All results are only transferable into other environments if formal design documentation of the required type (formal description of interfaces and algorithmic design) is used.

Open Questions

Research in this field usually tries to answer a few questions but results in creating more unanswered questions. Some of these unanswered questions are:

- What is an upper bound of module complexity - as measured by complexity metrics - causing no problems to deal with?
- How can the ratio between exterior and interior complexity (balancing aspect) be integrated in or added to metrics?
- How can the ratio between system complexity and average module complexity (balancing aspect) be integrated in or added to metrics?
- How can the answers for the balancing problems be used to determine something like an optimal design (relative to some requirements)?
- What aspects should be added to these ordinal metrics in order to obtain metrics interpretable with respect to interval scales?
- Can these results be transferred to different development environments?
- Do these results hold under realistic maintenance conditions?

References

[Basili 81]
Victor R. Basili, "Data Collection, Validation, and Analysis," in Tutorial on 'Models and Metrics for Software Management and Engineering, IEEE Catalog No. EHO-167-7, 1981, pp. 310-313.

[DeRemer, Kron 76]
F. DeRemer and H. H. Kron, "Programming-in-the-large versus Programming-in-the-Small," IEEE Transactions on SE, Vol. SE-2, No. 2, June 1976, pp. 80-86.

[Harrison et al. 82]
W. Harrison, K. Magel, R. Kluczny, A. DeKock, "Applying Software Complexity Metrics to Program Maintenance," IEEE Computer, Sept. 1982, pp. 65-79.

[Henry, Kafura 81]
S. Henry, D. Kafura, "Software Structure Metrics based on Information Flow," IEEE Transactions on SE, Vol. SE-7, No. 5, Sept. 1981, pp. 510-518.

[McCabe 76]
T. McCabe, "A Complexity Measure," IEEE Transactions on SE, Vol. SE-2, No. 6, Dec. 1976, pp. 308-320.

[Myers 75]
G. J. Myers, "Reliable Software through Composite Design," van Nostrand Reinhold Co.,

New York, 1975.

[Nehmer et al. 82]
J. Nehmer, R. Massar, W.-F. Racke, H. D. Rombach, R. Schrapel, "DISTOS - A Methodology to construct Distributed Operating Systems," Technical Report, Computer Science Department, University of Kaiserslautern, April 1982, in German.

[Rombach 84]
H. Dieter Rombach, "Quantitative Estimation of Software Quality Characteristics based on Structural Complexity," Ph.D. dissertation, Computer Science Department, University of Kaiserslautern, Fed. Rep. of Germany, 1984, in German.

[Rombach, Wegener 84]
H. Dieter Rombach, K. Wegener, "Experiences with a MIL design tool," Proc. 8th Conference on Programming Languages and Program Development, Zurich, Switzerland, March 1984.

Figure 1: Hierarchy of Design Descriptions

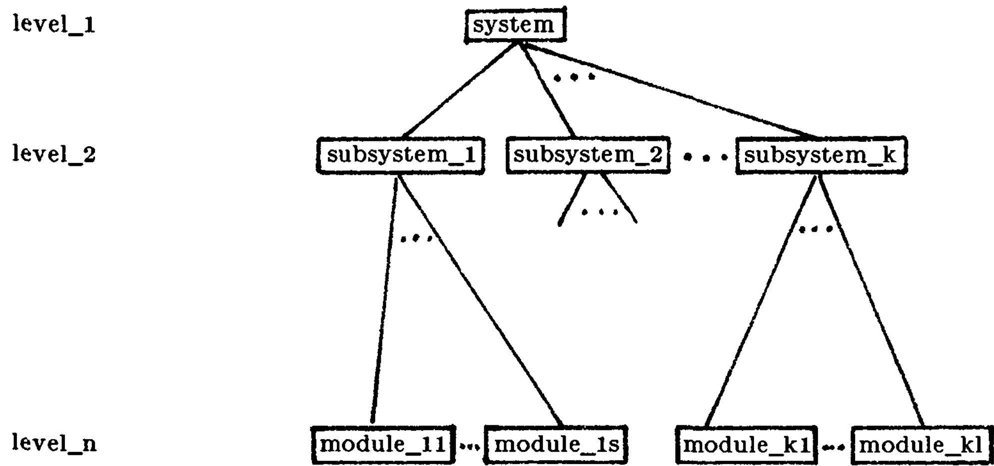


Figure 2: Example of a Module Design

```
MODULE  < name > :

SPECIFICATION:
   AUTHOR:  < name of designer >
   DEADLINE:  < date of completion >
   DATE:  < date of last change >
   VERSION:  < version number | to be increased with each
              'logical' change >
   PROBLEM:  < This part contains an informal (textual)
               description of the function of the whole
               module. Although it is informal, all information
               is to be ordered in a uniform way.
               Especially all the information, that can't be
               formalized in this stage of development, but
               already exists, should be documented in this
               section >
   EXPORT:    < All functions with parameters and types,
                offered by the module like:
                function_1 ( A : A_type, B : B_type ---> C : C_type) >
   IMPORT:  < All functions with parameters and types,
              used from other modules like:
              module_name.function_2 ( A : A_type ---> B : B_type )  >
END SPECIFICATION

IMPLEMENTATION:
   STATICS:  < The meaning, parameters, etc., of each
               exported function can be described in a
               informal way.
               These function specific parts are comparable
               to the PROBLEM part for the whole module
               in the SPECIFICATION above. >
   DYNAMICS:  < PASCAL-like description of a flow-graph.
               The nodes of this flow graph are
               nonlinear control flow operators like
               'if', 'while', 'for', 'case', etc.
               AND all accesses to the export
               or import interface. All edges (== linear
               parts without interface access) are represented
               by comments, later to be transfered into the
               code as comments. >
END IMPLEMENTATION

END MODULE < name >
```

Underlined key_words mark segments of formal representation of design information.

Table 1: Characteristics of developed systems

SW-Characteristics	Developed Systems					
	TSS_1	TSS_2	TSS_3	PCS_1	PCS_2	PCS_3
No.of MODULES						
- all objects/types	53/26	29/14	42/21	23/13	10/6	16/10
- processes (obj/types)	21/11	20/11	17/10	12/6	10/6	11/8
LINES OF CODE+	11000	10200	10800	1500	1460	1450
DEVELOPMENT EFFORT (in h)*	269.76	332.85	308.43	71.75	103.45	95.2
- system design	26.7	62.15	42.0	21.45	42.25	32.0
- module design	89.86	155.0	110.18	16.2	33.5	25.0
- system impl.	1.6	2.8	3.2	1.5	2.0	1.6
- module impl.	151.6	112.0	153.05	32.6	25.7	36.6
TEST EFFORT (in h)**	147.1	143.65	127.7	78.8	70.16	87.75
- module test	95.1	108.25	82.1	52.75	50.16	46.35
- system test	52.0	35.4	45.6	26.05	20.0	41.4
No.of CHANGES++	72	90	85	28	37	31
No.of LOGIC ERRORS	49	50	59	20	16	22
NO. OF CHANGES OCCURED IN PHASE (with average change effort (h))						
- system design	2 (10.2)	6 (5.1)	3 (7.0)	2 (9.5)	2 (7.15)	2 (6.1)
- module design	13 (3.1)	33 (1.8)	25 (1.2)	7 (3.2)	16 (2.12)	5 (2.8)
- system impl.	0 (-)	2 (2.0)	4 (0.9)	1 (2.1)	0 (-)	2 (4.2)
- module impl.	35 (0.8)	24 (0.92)	33 (0.78)	9 (1.2)	10 (1.6)	6 (2.2)
- module test	18 (1.2)	19 (1.6)	17 (1.26)	7 (1.38)	8 (1.5)	12 (1.05)
- system test	4 (3.5)	6 (2.1)	3 (6.05)	2 (3.8)	4 (2.6)	4 (4.0)
NO.OF CHANGES WITH EARLIEST DOCUMENT CHANGED						
- system design	11	22	14	10	3	3
- module design	24	28	22	7	16	11
- system impl.	0	2	3	1	0	1
- module impl.	37	38	46	10	18	16
AVERAGE No.of UNITS CHANGED PER REASON	3.04	1.9	2.47	3.14	1.81	2.43
- system	0.37	0.32	0.38	0.66	0.55	0.54
- modules	2.67	1.58	2.09	2.48	1.26	1.89

+ All lines except pure comment lines

* All development effort (no unit test) except time for compilation

** All test effort (module test, system test)

++ All types of changes except clerical errors

Table 2: STABILITY aspect

Spearman correlation coefficients between different types
of 'module complexity' and 'relative number of changed modules
per change cause during maintenance' respectively 'relative external
change effort in staff_hours per change cause during maintenance'
(separate for modules of two selected systems)

Types of Complexity		No. Changed Modules for		External Effort for	
exterior Compl.	interior Compl.	TSS_1	PCS_1	TSS_1	PCS_1
ISOLATED					
- control	**---**	**.5585+**	**.5494-**	**.6812**	**.6728***
- control	v (G)	.5020+	.4972-	.4819*	.4777-
- control	L	.5262+	.5182-	.4865*	.4807-
- control	**IA**	**.7222**	**.7128+**	**.6373**	**.6301***
- data	**---**	**.6126**	**.6084***	**.6718**	**.6658***
- data	v (G)	.5262+	.5182-	.4685*	.4612-
- data	L	.5376+	.5275-	.4710*	.4572-
- data	**IA**	**.7418**	**.7381+**	**.6537**	**.6510***
- information	**---**	**.6214**	**.6125+**	**.6704**	**.6643***
- information	v (G)	.5384+	.5290-	.4646*	.4599-
- information	L	.5510+	.5392-	.4689*	.4564-
- information	**IA**	**.7522**	**.7500+**	**.6498**	**.6473***
INTEGRATED					
- control	**---**	**.6440**	**.6382***	**.7363**	**.7270***
- control	v (G)	.6020+	.5933*	.5028+	.4982-
- control	L	.6102	.6071*	.5090+	.4998-
- control	**IA**	**.7736**	**.7602+**	**.7298**	**.7250***
- data	**---**	**.6458**	**.6412***	**.7780**	**.7729***
- data	v (G)	.6180	.6106*	.5401+	.5385-
- data	L	.6303	.6228*	.5454+	.5407-
- data	**IA**	**.7855**	**.7810+**	**.7709**	**.7684***
- information	**---**	**.7180**	**.7148+**	**.8065**	**.8005+**
- information	v (G)	.6412	.6364*	.5660+	.5609-
- information	L	.6584	.6507*	.5678+	.5633-
- information	**IA**	**.8200**	**.8168**	**.8008**	**.7978***
-----	v (G)	.4010*	.3801-	---	---
-----	L	.4609*	.4562-	---	---
-----	**IA**	**.6828**	**.6709***	**.6518**	**.6382***

- : significance >= .05, * : significance < .05, + : significance < .01,
otherwise : significance < .001

Table 3: MODIFIABILITY aspect

Spearman correlation coefficients between different types of 'module complexity' and 'relative internal change effort in staff_hours per change cause during maintenance' (separate for modules of four selected systems)

Types of Complexity		Relative Change Effort for			
exterior Compl.	interior Compl.	TSS_1	TSS_3	PCS_1	PCS_3
ISOLATED					
- control	---	.7870	.7718	.7268+	.7203*
- control	v (G)	.7005	.6922	.6846*	.6811*
- control	L	.7668	.7426	.6902+	.6872*
- control	IA	.5820+	.5765+	.5650*	.5601-
- data	---	.7352	.7308	.7014+	.7002*
- data	v (G)	.7021	.6931	.6690*	.6618*
- data	L	.7468	.7344	.6744*	.6688*
- data	IA	.5550+	.5488+	.5402-	.5365-
- information	---	.7554	.7498	.7112+	.7082*
- information	v (G)	.7216	.7155	.6740*	.6706*
- information	L	.7716	.7632	.6521*	.6480*
- information	IA	.6064	.6008+	.5253-	.5213-
INTEGRATED					
- control	---	.7221	.7172	.6925+	.6901*
- control	v (G)	.6918	.6808	.6704*	.6672*
- control	L	.7723	.7678	.6788*	.6734*
- control	IA	.6100	.6011+	.5512-	.5498-
- data	---	.6956	.6902	.7012+	.6999*
- data	v (G)	.7042	.7008	.6840*	.6790*
- data	L	.7962	.7916	.7024+	.6984*
- data	IA	.6244	.6196+	.5813*	.5776-
- information	---	.7312	.7265	.7318+	.7300*
- information	v (G)	.7496	.7440	.7182+	.7172*
- information	L	.8230	.8196	.7344+	.7289*
- information	IA	.6506	.6466+	.6071*	.6040-
-----	v (G)	.5619+	.5572+	.5846*	.5802-
-----	L	.7049	.7010	.6038*	.5965-
-----	IA	---	---	---	---

- : significance >= .05, * : significance < .05, + : significance < .01, otherwise : significance < .001

THE VIEWGRAPH MATERIALS

for the

H. ROMBACH PRESENTATION

121a

DESIGN METRICS
for
MAINTENANCE

H. Dieter Rombach *
Computer Science Department
University of Maryland

November 1984

* Research for this study was supported in part by the ministry of research and technology of the Federal Republic of Germany (DISTOS project).

OBJECTIVES

- Study the impact of system structure on software quality!

- Use design documents to measure system structure!

- Find design metrics

    - easy to automate

    - usable early

DESIGN DOCUMENTS

- The following design information is required in a formal, measurable way for each unit (system, subsystems, modules):

  - the functional interface of the unit (exported, imported functions)

  - internal realization of the unit functions (algorithm control flow)

- The concrete type of design documentation used in this study, is an extension of the module interconnection language (DeRemer & Kron):

  - Hierarchy of unit design documents

  - Each unit design document contains a

    * List of exported functions
    * list of imported functions

    * PDL-like description of control flow

GOAL - QUESTIONS

- GOAL: Determine the impact of structural software design characteristics on maintenance behavior!

The two maintenance aspects of interest are, according to two important activities:

- Stability <---> isolate
- Modifiability <---> change

- Question_1: Can software structure as available from design documents, be used to explain or predict STABILITY (number of changed units per change cause)?

- Question_2: Can software structure as available from design documents, be used to explain or predict MODIFIABILITY (change effort per unit and per change cause)?

SOFTWARE MODEL

- Number of algorithmic units (modules)
- Number of data structures
  - explicit
  - implicit

- Structure of each module is characterized by its

  - Exterior complexity
    (how the module is, or can be embedded
    in its environment)

    * control flow

    * data flow

    * information flow

  - Interior complexity
    (how the module functions are implemented)

    * control flow

    * Length

    * Intensity of interface access

EXPERIMENTAL APPROACH

OBJECTS:

- 3 Timesharing Systems (TSS)
  - $\sim$ 41 modules (20 module types)
  - $\sim$ 10,500 LoC

- 3 Process Control Systems (PCS)
  - $\sim$ 20 modules (10 module types)
  - $\sim$ 1,500 LoC

CONTROLLED MAINTENANCE EXPERIMENTS:

- 25 Failures
  - Faults identical for each system type
  - Fault types (control flow, data flow, data structure, interface, computation) with same distribution as during development
- 10 Environment Changes
- 15 Requirements Changes

SUBJECTS:

- 9 1-person teams
  - each team worked on one TSS and one PCS

DATA COLLECTION

- The following maintenance data were collected per change cause:

  - Number of changed modules
  - Effort in staff_hours per change cause to isolate faults
  - Effort in staff_hours per change cause to change

- The following structure data were collected per unit:

  - Number of exported functions
  - Number of parameters per exported functions
  - Number of imported functions
  - Number of parameters per imported functions

  - Number of exported functions with output parameter
  - Number of imported functions with output parameter

  - Number of exported implicit informations
  - Number of imported implicit informations

  - Number of units using exported functions
  - Number of units from which functions are imported

  - etc.

  - Number of independent paths - "v (G)"
  - Number of sequences without nonsequential control flow operator and interface access - "L"
  - Number of accesses (calls) to the unit interface - "IA"

DATA VALIDATION

- weekly meetings

DATA EVALUATION

- Spearman (R) correlation coefficients between

  - different types of module complexity

  and

  - number of changed modules per change cause (STABILITY)
  - effort in staff_hours per change cause (MODIFIABILITY)

- Hypothesis about relevant design metrics:

$$K \sim K_{exterior} * K_{interior}$$

  - Exterior Complexity
    - * Isolated Exterior Complexity
      Possible control flow
      Possible data flow
      Possible information flow
    - * Integrated Exterior Complexity
      Actual control flow
      Actual data flow
      Actual information flow
  - Interior Complexity
    Structure v (G)
    Length L
    Intensity of interface access IA

DATA ANALYSES RESULTS

- Question_1 (No. of changed modules ~ structure):

Spearman correlation coefficients (R) between "different types of module complexity" and "number of changed modules per change cause" for one representative system TSS_1:

Types of Complexity exterior	interior	TSS_1
ISOLATED		
control	---	.55+
control	v (G)	.50+
control	L	.52+
control	IA	.72
data	---	.61
data	v (G)	.52+
data	L	.53+
data	IA	.74
information	---	.62
information	v (G)	.53+
information	L	.55+
information	IA	.75
INTEGRATED		
control	---	.64
control	v (G)	.60+
control	L	.61
control	IA	.77
data	---	.64
data	v (G)	.61
data	L	.63
data	IA	.78
information	---	.71
information	v (G)	.64
information	L	.65
information	IA	.82
---	v (G)	.40*
---	L	.46*
---	IA	.68

*: significance < .05, +: significance < .01, otherwise: significance < .001

DATA ANALYSES RESULTS

QUESTION_1 (No. of changed modules per change cause ~ structure):

- Overall good correlations: 0.5 to 0.82
- Best correlation (0.82) for metric using
  - (integrated information flow, number of interface accesses IA)
- Sufficiently good correlations (0.78) for metric using
  - (integrated data flow, number of interface accesses)
- Sufficiently good correlations (0.64) for metric using
  - only integrated data flow
- Bad correlations (~ 0.4) for metrics using
  - only structure v(G), or
  - only length L
- Sufficiently good correlation (0.68) for metric using
  - only the number of interface accesses IA
- General correlation pattern:

$$(K_{exterior}, IA) > (K_{exterior}, \text{---}) > (K_{exterior}, v(G) or L)$$

DATA ANALYSES RESULTS

QUESTION_2 (Change effort per change cause $\sim$ structure):

- Overall good correlations: 0.6 to 0.82
- Best correlation (0.82) for metric using
  - (integrated information flow, length L)
- Sufficiently good correlations (0.79) for metric using
  - (integrated data flow, length L)
- Sufficiently good correlations (0.69) for metric using
  - only integrated data flow
- No correlations for metrics using
  - number of interface accesses IA
- Sufficiently good correlations (0.56 to 0.70) for metrics using
  - only length v (G), or
  - only structure L
- General correlation pattern:

$$(K_{exterior}, —) > (K_{exterior}, L) > (K_{exterior}, v(G)) > (K_{exterior}, IA)$$

PRACTICAL USE OF RESULTS

Use

- stability metrics of type
  (integrated data flow, IA)

- modifiability metrics of type
  (integrated data flow, L)

to

- decide between design alternatives
  (! ordinal !)

- plan testing effort

- check lower, upper bounds
  at different milestones

CONCLUSION

- It is possible to explain software maintenance behavior (MODIFIABILITY, STABILITY) by analyses of software structure as available from design documents.

- Best theoretical explanation:
  exterior complexity characterized by integrated information flow

- Best practical explanation:
  exterior complexity characterized by integrated data flow (EASY to AUTOMATE)

- Metrics without using any interior complexity show sufficiently good correlation.
  ==> VERY EARLY design documents (without any algorithmic design) can be used, to explain or predict maintenance behavior.

- These results are
  - drawn from formal design documents
  - validated by controlled experiments

- Unsolved problems are:

  - no sufficiently good linear regression between complexity metrics and maintenance data could be identified.

  - these results have to be validated for larger projects and realistic maintenance data.

  - the influence of the ratio 'exterior complexity/interior complexity' or 'system complexity/average module complexity' is not evident.

Db

N86-19973

An Approach to Operating System Testing

R. N. Sum, Jr.
R. H. Campbell
W. J. Kubitz
Department of Computer Science
University of Illinois
1304 W. Springfield Av.
Urbana, IL 61801

ABSTRACT

To ensure the reliability and performance of a new system, it must be verified or validated in some manner. Currently, testing is the only reasonable technique available for doing this. Part of this testing process is the high-level system test. This paper considers system testing with respect to operating systems and in particular UNIX. This consideration results in the development and presentation of a good method for performing the system test. The method includes derivations from the system specifications and ideas for management of the system testing project. Results of applying the method to the IBM System/9000 XENIX operating system test and the development of a UNIX test suite are presented.

1. Introduction

Every new system must be evaluated before delivery to ensure proper functioning and reliability. One part of this evaluation is the system test. A system test validates the high-level functionality of a system. In the context of a general purpose computer operating system, a system test verifies that the user interfaces conform to the system's specifications.

Verification, in the form of program proofs, would eliminate the need for system testing. However, the current proof techniques are not yet adequate. Therefore, a systematic approach to system testing is needed. This paper describes a heuristic approach to a software system test that derives its tests from the system specifications. The approach includes individual tests embedded in a comprehensive testing framework. This paper describes the application of this approach to the system test of a XENIX® operating system for the IBM Instruments, Inc. System 9000.

1.1. System Test Overview

The goal of system testing is to show that a system does not meet its specifications [Myers79]. For a system test, two classes of specification must be considered. The first class is provided by an overview of the components of the system which is often called the "formal system specification" [Beeru83]. The second class is the user documentation which includes user's manuals, operator manuals, and hardware manuals [IBMIn84a, IBMIn84b, IBMIn84c]. With these two classes of specification, a hierarchical testing framework can be designed in a top down manner for the system test.

The list of user interfaces provided by the formal system specification can be used to organize the test suite into a hierarchical framework. Each listed interface is tested by a corresponding component of the test suite. Although this approach may not be appropriate

XENIX *is a trademark of Microsoft*

for all systems, for UNIX® we found it provided a natural decomposition of the test suite. Each component can take advantage of particular properties of the system interface (for example, whether the interface is programmable or is interactive) while the decomposition organizes the particular testing methods and ensures that all the interfaces are tested.

Here we describe the decomposition of the test suite into components, the specific testing techniques used in the components, and the results of applying the test suite. We present several testing strategies which were required because of the particular properties of the user interface. We then summarize the errors discovered in the system test and the manpower effort required to generate the test results. The XENIX system tested is a port of a commercially available software system and required the programming of new device drivers and machine dependent code. In our conclusions, we attempt to identify to what extent the errors we discovered could be associated with the components of the system which were rewritten for the port. Our analysis reveals that there is only a small correlation between the rewritten software and the errors that were identified.

2. Development of a UNIX Test Suite

The XENIX operating system test suite is organized according to the structure of the interfaces specified in the system specification. Examples of tests in the System 9000 test suite will be used to exemplify our testing methods and methodology.

2.1. The Test Suite Structure

The formal specification of the System 9000 described four major user interfaces and these were adopted as the major components of the test suite. They are:

1. Commands – the high-level commands available to every user,
2. Subroutines – the subroutine libraries designed for use in application programs,

UNIX is a trademark of Bell Laboratories

3. System Calls – the subroutines designed for use in systems programs that directly invoke operating system functions,

4. Device Drivers – the interface designed for use in systems programs that request access to hardware devices attached to the System 9000.

The four interfaces provided a useful global organization for the test suite. Each component of the test suite had different testing concerns and required different testing techniques

2.2. The Components

Each component of the test suite includes programs, test plans, and documentation for each function to be tested. The design of the tests is based upon the usage specified by the manuals that make up the second class of system specifications. The major issues that arise in the design of a test involve *test style* and *test coverage*.

Test style refers to the manner in which the test is performed. There are three approaches used in the test suite: *interactive procedures*, *guided programs*, and *automated programs*. The test style selected for a test is determined by the properties of the interface being tested. For example, many interactive user commands are tested by a user following an interactive procedure which yields reproducible results. However, most programming libraries are tested using automated programs.

Test coverage concerns the development of a sufficient number of tests to ensure that all of the functions provided by an interface are tested. The manuals describe the functions provided by the manual's interface and how they are expected to interact. For example, coverage of a math subroutine library includes determining that all the functions exist, and that they take the specified number and types of parameters. Notice that coverage for a system test may differ from coverage for a function test that is used to test the isolated function before system integration. Although it is desirable to test every valid parameter, this is often too time consuming in a system test and would duplicate work performed in the

function test.

Due to the variety of interfaces contained in the system, each test suite component employs different test styles, test descriptions (documentation), and test derivations. Some of the test styles, descriptions, and derivations encountered in the UNIX test suite are described below.

2.2.1. Test Styles

As a consequence of the form of the interfaces in UNIX, the most common testing styles used in the test suite are interactive procedures, guided programs, and automated programs. When testing an interactive environment, intuition suggests the use of an *interactive procedure* style. For example, text editors such as "vi" provide an interactive environment in which user commands are executed. An interactive procedure may be the only means to ensure easily that the editor commands correctly update the screen of a terminal. Consequently, most of the tests in the commands sub-suite are interactive procedures. A *guided program* is a small, interactive, test program and is a hybrid of an interactive procedure and an automated program. Guided programs are used if an interactive procedure is undesirable or tedious but the expected response cannot be easily calculated within an automated program. For example, guided program tests are used in the test suite for high level terminal input/output subroutine packages such as "termlib", "termcap", and "curses" and allow a person to examine the effect of a long sequence of operations on the display of a terminal by comparing the resulting output with a standard pattern. An *automated program* is used when a program can easily calculate the expected response and check its correctness. This approach was taken for most low level programming interfaces including system calls.

The different forms of user interface also produce different forms of test descriptions as well as test styles.

2.2.2. Test Descriptions

Corresponding to each testing style is a particular form of low level documentation that describes the test and its execution. The documentation used in the test suite for automated program tests consists of a standardized header that is prepended to the program code and describes the test, its use, and any dependencies. When a group of related tests (for example, the system calls) all use automated programs, a common logging system is used to record the test results. Separate documentation for the logging system is provided. The documentation for guided programs and interactive procedures must supply a precise script for the user as well as describing the test. A *Test Definition Form* (see Appendix) is used in the test suite to supply this information. The *Test Definition Form* contains a standardized header that is similar to the one used for automated programs and a procedures section containing an enumeration of the commands to perform and the responses to expect. Although the *Test Definition Form* is not quite as exact as a program, it provides a means of defining reproducible tests that could be reliably executed by any member of the testing team.

2.2.3. Test Derivation

Individual tests in the test suite are designed by studying the manuals relevant to each function. The number of tests and the test data for each function are dependent upon the size and complexity of the function and include exception testing and stress testing. *Exception testing* involves the erroneous use of the function and the subsequent error handling used by the system. *Stress testing* explores whether the system will support the extremes specified in its documentation.

In most cases, because the manuals are written in imprecise English, the mechanical derivation of test data for a system test is impossible. This placed most of the burden for choosing test data for the test suite on the individual test developer. A test data design methodology was developed that involved examining the input and output specifications

found in the manuals and applying a set of simple test data derivation techniques including:

1. Exhaustive testing – the use of *every* possible data value,
2. Random testing – the use of values chosen randomly throughout all of the input ranges of the function being tested,
3. Special case testing – the use of particular values that are chosen because they exercise the function at the limits of its range and domain,
4. Explicit case testing – the use of values explicitly used or suggested in the manuals.

In general, functions with a very small (less than 10) input range were tested exhaustively while functions with a larger input range used a composite of random, special, and explicit data values. The "abort routine" of UNIX is an example of a function that is easy to test exhaustively because it requires no parameters. The system call "write" is an example of a function that has many possible parameters (including file descriptors and buffer interfaces) for which a composite test data derivation technique is appropriate.

Exception tests and stress tests are employed in the test suite whenever possible. Exception tests included test data for error conditions described in the user manuals for which error handling was defined as well as test data that would obviously correspond to an error but would not correspond to a documented error condition. An example of an exception test is a program that writes to a file that is open for read only access. Stress tests were applied to determine system response when its limits were reached. Stress tests were used to exercise device drivers, memory management routines, and file allocation and deallocation. Test data used in the stress tests included requests involving maximum program and file sizes. One example of a stress test used in the test suite is a program that requests as much memory as the system has available.

3. Results of Testing the System 9000 XENIX

We now describe some of the results obtained in applying the test suite developed above to the system test of a pre-released version of the System 9000 XENIX operating system. Most of the bugs that were discovered in the system test described have since been fixed or documented as restrictions in the user manuals. A terse description of the system is followed by some general results and discussion of problems of particular interest. Each bug found was documented in a *Problem Tracking Memorandum* (PTM). The documentation describes the error found and the likely software component that contains the software fault that generated the error.

3.1. System 9000 XENIX

The System 9000 is a small MC68000-based system designed for use as a workstation. The System 9000 XENIX operating system is a port of the Microsoft's Version 7 UNIX-based XENIX operating system and supports multiple users and processes. The entire operating system occupies approximately 7Mbytes of hard disk storage including all binaries and system data files. The memory resident part of the system occupies approximately 144Kbytes of memory. The source code of XENIX is proprietary and was not available to the test team. This has made it difficult to estimate the number of faults found relative to the number of lines of code tested.

3.2. General Results

Table I, *PTMs by Test Area*, shows how the software system faults are distributed within the interfaces and specifications (user documentation.) Without any detailed knowledge of the implementation of XENIX on the System 9000, one might expect the largest number of bugs to be present in the commands since these represent the largest amount of code. However, the System 9000 has a ported operating system and based on this knowledge, one might expect that a greater number of software faults would be found in the

Table I.
PTMs by Test Area

Test Area	Number	Percentage
Commands	81	51.92
Drivers	5	3.21
System Calls	24	15.38
Subroutines	15	9.62
Specifications	29	18.59
Incomplete Data	2	1.28

device drivers and machine specific parts of the operating system. Most faults were found within the commands.

The proportion of the bugs found in the system calls is more serious than other bugs because these reflect failures in direct requests for operating system services. These bugs and hardware bugs occasionally interrupted the system test schedule while they were fixed. Documentation errors were expected since both the documents and the system were developed simultaneously. (The *Incomplete Data* category indicates that a couple of PTM forms were not filled in completely.)

Table II, *PTMs by Test Type*, is a summary of where the bugs where found based on whether the test tested normal usage or error-handling (exception testing). While exception testing in general did not display any unusual trends, it did discover significant system bugs, particularly in the system calls and subroutines.

Table III, the *Severity Level Summary*, compares the number of bugs against their impact on the system. Severity level 1 bugs are the most severe and cause the system to

Table II.
PTMs by Test Type

Test Type	Number	Percentage
Exception	17	10.90
Normal	137	87.82
Incomplete Data	2	1.28

Table III.
Severity Level Summary

Level	Number	Percentage
1	10	6.41
2	22	14.10
3	90	57.69
4	34	21.79

crash no matter how the command is used. Level 2 bugs cause a function not to work or a part of a function to crash the system. Level 3 bugs cause part-of-a-function not to work. Level 4 bugs are documentation errors or cause an "annoyance" form of error. Based on the description of the severity levels, it is not surprising that level 3 has the biggest percentage of the total. The distribution of bugs appears to be what one would expect in a software system test.

Table IV, *Univ. of Illinois Man-Power Summary*, corresponds to a typical curve for project manpower usage. The productivity in the early months reflect the design of the test suite. (Some delay in November was incurred due to a problem with shipping the systems.) In January 1984, the bulk of the tests were coded and the number of bugs discovered peaked. Finally, as the number of test cases increased, the bugs found decreased, and the manpower devoted to testing was reduced. It was found that several of the test cases were difficult to formalize and code. Because of the desirability of generating results, the difficult

Table IV.
University of Illinois Manpower Summary

Month	RA-months	Number of PTMs
Oct. 83	1.5	0
Nov.	3.0	0
Dec.	3 0	14
Jan. 84	6 0	23
Feb.	6.0	19
Mar.	6.0	19
Apr.	3 0	6
May	2 0	2

tests were often deferred until later in the testing period. This also contributed to the decline in reported bugs since the difficult tests took longer to design and code. (One RA-month is approximately one-half of a man-month.)

3.3. Further Analysis

The test results revealed several interesting faults related to the hardware, the C compiler, the file system, and commands.

Nine bugs were found in the hardware during the software system test. Many of these were discovered as a result of the stress put on the hardware system by the software system testing activity. It was somewhat unexpected to have the software test discover some timing errors in the hardware.

The C compiler was found to have at least three bugs directly traceable to the origins of an early portable C compiler that was several years old. This was discovered through previous testing of other C compilers at the University. At first, a complete test of the C compiler could not be accomplished because it would not compile the C test suite programs. Based on the earlier testing of other C compilers, a list of suspected as well as confirmed bugs was dispatched to the developers. Because the operating system is written in C, the defects in the C compiler are potentially very serious. However, we believe that a cross compiler was used for the port, not the system's C compiler. We were unable to test the C compiler used for the port but suspect that it too may have contained some bugs that showed up as faults in system software.

Table V, *Faults Distributed by Function,* displays the distribution of faults over the various subsystems within XENIX. File system bugs when collected across the various interfaces amounted to 15 per cent of all bugs found. This could mean that many of the file system bugs were dependent on a few of the device driver bugs or that there may have been some latent design flaws remaining in the file system. Unfortunately, we have too little

Table V.
Faults Distributed by Function

Function	Number	Percentage
File System	23	14.74
Hardware	9	5.76
C Compiler	5	3.2
Memory Mngmt.	2	1.28
Other Kernel	6	3.84
Other Software	111	71.1

information to allow us to draw a conclusion.

Twenty three of the bugs we discovered (the C compiler's bugs were counted as one bug for this purpose) appeared to be attributable to the XENIX system rather than to the port. These bugs included a read and write system call that failed to provide appropriate error handling when invoked with a null buffer pointer parameter. A stress test also revealed that the file system would allow more links to be made to a file than the documented limit. In this case, the documentation was correct and the error checking within XENIX was inadequate.

Finally, a couple of command bugs proved rather disquieting. The first occurred in the system shutdown command and caused the system to hang rather than clean itself up correctly. The second, which caused much amusement, occurred in the XENIX "remove user" command and always caused the removal of every user in the system as well as the requested user. This obviously rendered the system unusable after everyone logged out.

4. System Test Management

This section describes some of the management aids that were used during the development of the test suite and during testing. The aids supported test development, reporting bugs, organizing man power, and providing maintenance tests.

4.1. Guiding Test Development

To guide the system test, a system test plan [Morri83] was drawn up by a small test design team. This system test plan included objectives for the system test, outlines of the testing to be done, naming conventions for the tests, and a very loose estimate of the size of the project. Essentially, the system test plan was an informal requirements and specification document for the test suite.

To ensure system test coverage in the components of the test suite, a set of matrices was used that cross-listed the proposed tests with the functions to be tested. At least one matrix was used for each interface and some large interfaces required several matrices. Although these matrices tended to be sparse, they did provide a convenient way to check coverage of the tests. For the final presentation in the test suite [Sum84] documentation, the matrices were compressed into a more compact tabular form.

4.2. Reporting Bugs

To manage bug reports, a *Problem Tracking Memorandum* (PTM) was used. The PTM form included information about its originator, place of origin, severity level, date of origin, test number, the operating system release, the hardware configuration, and both a short synopsis and detailed description of the problem. (A sample PTM is included in the Appendix.) These forms were filled out by a test team member upon discovery of a bug. The forms were then relayed from the test team to the developers responsible for the problem area. After the bugs were fixed, a response to the PTM was returned and the test repeated. If the retest was successful, the PTM was closed, that is, the bug was considered fixed.

4.3. Organizing Manpower

The allocation of manpower to system test development appears in retrospect to have followed a similar pattern to software development. This is attributable to the close

similarity to the processes involved. Initially, a few people were assigned to design and develop the test plan. After testing began, people were added to develop and code tests and to execute the tests. Finally, as the discovery of new bugs decreased and the test suite neared completion, the number of people was decreased.

4.4. Communications

A problem that was solved during the project was a method for exchanging PTMs (bug reports) between the test team and development group at IBM Instruments in Danbury, Connecticut and the test team at the University of Illinois. This problem was solved by using a dedicated *notesfile* [Essic82, Essic84] on one of the University of Illinois computers. A notesfile is a news/bulletin board system that allows notes and responses to be appropriately grouped and managed on-line. The PTM notesfile was checked daily by the team in Danbury by logging in over long distance phone lines. This form of communication proved to be faster and more effective than using the U. S. mail service or reporting the bugs by telephone. The scheme resulted in an rapid exchange of bugs and fixes and permitted quick qualification of ambiguous descriptions in the PTMs.

4.5. Maintenance Provisions

A key reason for developing the test suite is to help facilitate regression testing of future operating system releases. The test suite organization was instrumental in providing this ability. The hierarchical framework used for the test suite, together with the UNIX file system, provided an easy way to store the test suite on line. The UNIX text processing facilities encouraged full documentation.

5. Conclusion

Construction of the test suite was, we believe, valuable to both IBM and to the students who participated in the project. We believe that twenty three bugs originated in the

XENIX software. This demonstrated the benefits of a systematic system test. The large number of bugs discovered in the XENIX commands compared with the small number discovered by testing the device drivers, memory management, and system calls would appear to suggest that many of the machine dependent, port generated faults could be more easily discovered in a system test than in an isolated function test of a system component. However, we have been unable to verify this for lack of information concerning the nature of the fixes that were made to the system and for lack of access to the source of XENIX. We were intrigued that we could identify the C compiler used in the System 9000 XENIX by the errors it contained and somewhat dismayed at how such errors could be tolerated in commercial software for such long periods of time. Finally, although the construction of the test suite was tedious at times, it did provide a significant learning experience for the students and many of them have continued on to become very knowledgeable UNIX users.

6. Acknowledgements

The authors wish to acknowledge the help and cooperation of the entire Professional Workstation Research Group of the University of Illinois at Urbana-Champaign. Also appreciated was the cooperation of John Morris and his staff at IBM Instruments, Inc. in Danbury, CT and the funding from IBM which made the project possible.

7. References

[Beeru83][1] Beerup, Carl, *CS 9000 XENIX System Programming Functional Specification*, IBM Instruments, Inc., November 1983.

[Essic82] Essick, Raymond B. IV and Rob Kolstad, *Notesfile Reference Manual*, Technical Report UIUCDCS-R-82-1081, 1982.

[Essic84] Essick, Raymond, B., *Notesfiles*, M.S. Thesis, Technical Report, UIUCDCS-R-84-1165, 1984.

[1] This document is internal to IBM and not available to the general public

[IBMIn84a] IBM Instruments, Inc., *XENIX System Device Driver Manual*, March 1984.

[IBMIn84b] IBM Instruments, Inc., *XENIX System Operations Manual*, March 1984.

[IBMIn84c] IBM Instruments, Inc., *XENIX System Reference Manual*, March 1984.

[Morri83][2] Morris, John D., *CS 9000 XENIX System Test Plan*, IBM Instruments, Inc., November 1983.

[Myers79] Myers, Glenford J., *The Art of Software Testing*, John Wiley & Sons, Inc., 1979.

[Sum84] Sum, Robert N., Jr., et al., *UNIX/XENIX Test Suite - IBM S9000 System Test*, Report of the Professional Workstation Research Group, Dept. of Computer Science, University of Illinois, June 1984.

[2]This document is internal to IBM and not available to the general public

8. Appendix: Test Definition Form

This is a sample *Test Definition Form* as used for defining and executing the interactive and guided program tests.

Test Definition Form

Testcase Id: UXCMD103 **Author:** Robert Sum
Date Written: 2/9/84
Modified By: Robert Sum **Date:** 2/15/84

Function:

Mkuser is the usual way to add users to the XENIX system.

Description:

Use mkuser to create a new user for the system.

Dependencies:

Tester must be a super-user.

Restrictions:

Copyright (C) 1984
Robert Sum
IBM Workstation Research Project
Department of Computer Science
University of Illinois

Procedure:

1. Enter-

a) do: mkuser

b) when prompted enter 'mktester' as the user name.

c) when prompted enter 'mkpasswd' as the user passwd.

d) when prompted enter 'Make User Test' as the user comment.

e) when asked if everything is ok, check it out and respond accordingly.

f) when everything is ok, answer affirmatively and wait.

System Response

a) The program will pause for you to check once more.

b) Then it will create the user passwd file entry, home directory, mail file, his introductory mail, and his '.profile' file.

2. Enter-

a) do: more /etc/default/mkuser

b) Remember the default home directory and shell.

c) Change directory to the home directory, i.e. 'default home directory'/mktester.

d) do: l

e) do: cmp .profile /usr/lib/mkuser.prof

f) do: more /usr/lib/mkuser.mail

System Response

a) Changing directory should act silently.

b) l should list just the profile.

c) cmp should not return anything, i.e run silently.

d) Remember what the mail is.

3. Enter-

a) do: logout

b) Login in as mktester.

c) do: printenv

d) do: more .profile

e) do: mail

f) do: q

g) do: logout

System Response

a) Login should be successful.

b) Result of printenv should agree with things set in '.profile'. NOTE: This is true only if the default shell is the Bourne shell (sh).

c) mail should mail the output of the more in part 2 with an added header.

d) q just exits mail.

e) Logout should be successful.

Comments:

This test should be chained with UXCMD105 which tests rmuser.

9. Appendix: Problem Tracking Memorandum

This is a sample *Problem Tracking Memorandum* as used to report bugs during the testing. It is filled in as if the bug has just been discovered.

Problem Tracking Memorandum

SYS-2003

Severity Level: 2
Problem Summary: C compiler error: expression causes compiler loop.
Originator: R. Sum **Department**: UIPWG **Extension**: (217)333-8741
Regression Test:

Opened: 12/1/83 **Answered**: / / **Verified**: / / **Closed**: / /
Test Case Number: UXCMD801

Publication Title: N.A. **Draft Date**: / /
Software Level: Driver 2**Hardware Level**: N.A.**Application Level**: N.A.

Problem Description: This compiler error message is generated by moderately long expressions, particularly when doing some type casting. The following generated the error:

```
if( (int)c != 26 || (int)s != 26
 || (int)l != 26 || (int)u != 26
 || (int)f != 26 || (int)d != 26 ) lrc = lrc+4;
```

where

c is a char variable,

s short

l long

u unsigned

f float

d double.

lrc is an integer local return code.

THE VIEWGRAPH MATERIALS

for the

R. SUM PRESENTATION FOLLOW

An Approach
to
Operating System Testing

Robert N. Sum, Jr
Roy H. Campbell
William J. Kubitz

Department of Computer Science
University of Illinois
Urbana-Champaign

28 November 1984

System Test Objective

Goal: Show that the system does not meet its specifications.

Ideal: Program and System Proofs, but ...

Reality: Use a *Good* Heuristic Approach

Example: System/9000 System Test

System Interfaces: High-Level Testing Structure

Idea: Decompose the system into its user interfaces and test each one.

Example: XENIX on the System/9000 has:

1. Commands – day-to-day user commands
2. Subroutines – high-level programming
3. Systemcalls – low-level system programming
4. Drivers – low-level hardware programming

Test Styles

- Interactive Procedures

- Guided Programs

- Automated Programs

Test Derivations

- Exhaustive Testing
- Random Testing
- Special Case Testing
- Explicit Testing
- Exception Testing

Test Results

PTMs by Test Area		
Test Area	Number	Percentage
Commands	81	51.92
Drivers	5	3.21
Systemcalls	24	15.38
Subroutines	15	9.62
Specifications	29	18.59
Incomplete Data	2	1.28

Test Results

Severity Level Summary		
Level	Number	Percentage
1	10	6.41
2	22	14.10
3	90	57.69
4	34	21.79

Description of Severity Levels:

1 Very Severe - function causes system crash

2 Severe - part of a function does not work, may cause crash

3 Usual - part of a function does not work, little system impact

4 Annoyances - Documentation and micellaneous minor errors

Test Results

Faults Distributed by Function		
Function	Number	Percentage
File System	23	14.74
Hardware	9	5.76
C Compiler	5	3.2
Memory Mngmt.	2	1.28
Other Kernel	6	3.84
Other Software	111	71.1

Test Results

Univ. of Illinois Man-Power Summary		
Month	RA-months	Number of PTMs
Oct. 83	1.5	0
Nov.	3.0	0
Dec.	3.0	14
Jan. 84	6.0	23
Feb.	6.0	19
Mar.	6.0	19
Apr.	3.0	6
May	2.0	2

Management Problems

Error Tracking

Man-power Allocation

Coordination of Test Teams

Regression Testing

Problem Tracking Memorandum (PTM) form used to keep all information together.

Man-power followed a *standard* project team method with increases and decreases as testing proceded.

Distance between in-house and out-of-house test teams was bridged by keeping PTMs on line and having the in-house group check it at least daily.

The Tests were organized into the *UNIX/XENIX Test Suite* which includes the code and documentation for running the tests.

Conclusions

- The Value of System Tests.
- The Difficulty of Fault Location.
- Fingerprinting Software by its Bugs.
- Bug Survivability.
- Testing as a Learning Experience.

N86-19974
D7

The Cognitive Connection: Software Maintenance and Documentation[1]

Elliot Soloway*
Stan Letovsky*
Beatrice Loerinc**
Art Zygielbaum***

*Department of Computer Science
Yale University
New Haven, Connecticut 06520

**Department of Statistics and Computer Information Systems
Baruch College - CUNY
New York, New York 10010

***Jet Propulsion Laboratory
California Institute of Technology
Pasadena, Calif.

Abstract

With the goal of trying to understand what software maintainers do, we conducted talking aloud, video-taped protocols with four expert maintainers as they were actively engaged in the process of enhancing a relatively small, interactive database program. Our subjects exhibited a number of different types of information gathering strategies. Underlying these patterns of behavior, however, was the use of *expectations* about what should be seen in the program under examination. These expectations were generated on the basis of knowledge previously acquired as to the the goals and programming plans that are typically employed in realizing interactive database programs. Thus, while the experts seemed to possess adequate programming knowledge, their actual code patches violated a basic principle of program structure. We attribute this failure by the programmers, at least in part, to ineffective program documentation. We conclude with suggestions for changes in the content of program documentation that should better facilitate software maintenance.

1. Introduction: Motivation and Goals

Our collective consciousnesses are in the process of being raised to the important problem of software maintenance: it is clear that program maintainers need new tools to aid them in their significant chore. The approach we take to the development of such tools is one that we have

[1]Research described in this paper was carried out in part at the the Jet Propulsion Laboratory, and in addition was supported by the Jet Propulsion Laboratory, California Institute of Technology under contract with the National Aeronautics and Space Administration.

taken in a number of other similar software engineering situations [2]:

> we first try to understand how the maintainer does (and fails to do) the task of program maintenance; we are then in a better position to suggest tools/methods that can aid him in the specific areas in which he is having the most difficulty.

Towards this end we have carried out a video-taped study with actual program maintainers at JPL. In this paper we present first, some observations of what the maintainers did --- and most importantly, *did not* do --- and second, recommendations for changing the content of software documentation that we feel should facilitate the maintainers doing a better job of maintaining software. We hasten to point out the work reported here is only a beginning: the conjectures we make based on this work cry out for further experimental studies, which we in fact plan to carry out. Nonetheless, we feel the results gathered so far are already intriguing enough to justify presentation.

2. Details of the Study

We video-taped 6 professional programmers "talking aloud" as they were engaged in the task of adding a new feature to an existing program. The talking aloud methodology allows us to better view the process of software maintenance; this type of data is an important source from which to develop a cognitive theory of software maintenance. Subjects in our study were 4 expert level program maintainers and 2 junior level program maintainers;[2] the former had between 3 and 20 years of professional programming experience, while the latter had less than 3 years of professional experience.

We presented each of the subjects with a Fortran 77 program that managed a small, interactive database of personnel information, henceforth referred to as the PDB program. The program contained 15 routines, for a total of approximately 500 lines of code. Figure 2-1 presents an overview of this system. In fact, this exact overview was provided to the subjects as part of the documentation of the program. In addition to the brief *Overview*, the documentation contained the following (in this order):

- *Program Module Descriptions:* each module was described in terms of its specific function and its use of variables;
- *Hierarchy Chart:* the calling structure of the modules was given;
- *File Description:* the structure of database file was given;
- *Sample Session:* a trace of the use of the PDB was given.

Our intention was to make the documentation of the PDB reflect generic standards for program documentation.

[2]In this paper we will not analyze in detail the behavior of the junior level subjects; rather we will focus on the experts.

The personnel data base system provides online personnel information. As the sample session below illustrates, the user can issue various commands to view or make modifications to the entries in the database. SHOW allows the user to see the contents of an already existing record. CREATE allows the user to create a new record. DELETE will delete an existing record and UPDATE will allow any field of the record to be changed. A session ends when the user issues the EXIT command.

Figure 2-1: Personnel Data Base System: Overview

The Personnel Data Base System provides online personnel information. Today, we ask you to increase the functional capability of this system by making the following enhancement:

> Allow the user to restore a record that was deleted during the current session. For example, assume that the user deleted the following record during a session with the Personnel Data Base System:
>
> ```
> Soloway,Elliot,M
> 177 Howard Ave
> New Haven, Ct   06519
> 203 562-4151
> Dunham Labs 322C
> 436-0606
> ```
>
> Deleting a record makes that record unavailable for subsequent access. The enhancement we are asking you to make would allow the user to restore a deleted record to the data base, during the same session that it was deleted it. For example, a user who had deleted the above record could then restore it during the same session. The record is thus returned to active status and is available for subsequent access.

Figure 2-2: Enhancement Task

Figure 2-2 describes the enhancement task that our subjects were asked to perform. Briefly, they were asked to add a function to the PDB that would allow users to restore a record that was deleted in the current session. Three of the 6 subjects completed the task in the allotted 90 minutes.

3. Recurrent Behaviors

While there was considerable variability in the details of how our subjects performed, we were still quite able to abstract a number of behaviors that essentially all of our expert subjects exhibited. In what follows we identify and describe these key strategies.

- *Model-directed program understanding:* While the experts had apparently never designed a program exactly like the one we gave them to modify, they nonetheless had considerable experience with programs similar to the PDB. Not surprisingly, the expert subjects employed this experience in coming to understand the given program. In particular, experts were continually drawing on their knowledge of similar systems to set up *expectations* about what they should see in the program at hand. These expectations guided subsequent program analysis.

  The expectations formed and used by our subjects dealt with identifying the *goals* and *programming plans* in the code. That is, our subjects drew on their knowledge of

database systems in general in order to predict that certain *goals* would need to be achieved in the program in order to achieve the higher level objectives stated in the *Overview*. Moreover, our subjects drew on their knowledge of generic programming techniques --- which we have called ***programming plans*** --- in order to predict the manner in which the goals would be realized. Previously, we have presented arguments, plus supporting empirical data, that programmers do in fact have and use this type of knowledge in comprehending programs [1].

For example, the quotes given below, taken from the video-taped protocols with the experts that, illustrate these claims. In the first quote, one expert assumes that the routine called GETDB will accomplish the goal of inputting the database:

```
Subject: ... Ok. It would call GETDB.
        We don't know what that is yet --
        we won't worry about that.

Experimenter: Ok. You're not going to worry about that?

Subject: Well, I'm going to assume that
         it gets the file into memory.
```

In the next quote, we see an expert predicting the standard, alternative ways that a database array will be searched for a record key:

Experimenter: So what does this tell you?
What are you thinking about?

Subject: ... Just trying to figure out
how you step down [*through the array*].
If this thing is by number
or by last name or how it's
basically indexed in the array.
They use pointers I suppose.

In the following quote, we see an expert making a prediction and then going to the code to verify that prediction.

Subject: Ok. I'm down to GETDB here [in the code].
Now, the subject turned back to GETDB in the documentation.

Experimenter: Why?

Subject: Just to make sure that what I understand
it to do here is the
same thing as it says it's going to do there.
And if not; why not.

We call this information gathering strategy *model-directed* since the experts were employing an abstract characterization of database programs, expressed in terms of goals and plans, to direct the process of understanding the specific database program given to them. As described more fully below, the model-directed strategy was used

in two different ways by the experts.

- *Systematic Perusal of the Program:* Several of our experts spent considerable time (approximately 35% of their 90 minutes) trying to understand much of the PDB program *before* they attempted to carry out the specific modification. They used, of course, expectations to guide their understanding; however, they were attempting to understand more about the program than would *seemingly* be required in order to correctly make the desired enhancement. We have several possible interpretations of this behavior:

  1. the subjects who employed this strategy explicitly voiced their concern that undocumented interactions between parts of the program that could impact in some way on their subsequent enhancement
  2. while the PDB program was written using, what we believe to be, standard programming plans and rules of programming discourse, subjects may have been less confident that the program was in fact going to conform to their expectations; in other words, programmers may not have trusted the program to be written in a standard manner -- i.e., one that would be in accord with their expectations.

  It is entirely reasonable to suppose that in fact both interpretations are correct.

- *As-needed information gathering strategy:* Several subjects did not employ a systematic strategy, but rather after a very brief examination of the program and documentation, started right in on the actual enhancement. However, questions arose about aspects of the PDB program that they needed to know -- which they didn't then know -- in order to insure that their enhancement would indeed fit correctly in the existing program. In these situations, subjects would then go back to the program and to the documentation in order to find answers to these questions. It is important to note that by and large the searching for information was very focused; there were no real fishing expeditions. Rather, guided by their expectations, again, they were to able to pose specific questions about what they needed to know, and they were able to predict where in the code answers to those questions were to be found.

  For example, in the quote given below taken from one expert we see him deciding to look back at the code in order to answer a specific question. Notice that the expert had already begun the modification.

```
Subject:   I'd say I've got the big picture on what it
           [the program]
           does.  ... Of course, there's some of these
           searching mechanisms I
           haven't looked at that but...
           That may be complex, I don't know.  I don't
           really care.

Experimenter:  Why not?

Subject:   Ah.  Well, to do this one [this modification]
           --it assumes that
           this thing can go out and search.
           Although I probably should look to
           see...Good thought.  Maybe it won't find a
```

```
record that's deleted.  I'll
take a look at SEARCH.  Ok.
```

It is interesting to speculate as to why subjects would employ this latter strategy. One suggestion, consistent with our observation that expectations played a key role is this: after the brief perusal of the code, subjects found that the code met their expectations, and thus became confident that the rest of the code would also meet their expectations. In other words subjects employing the *as-needed strategy* felt that there would be no surprises, and that therefore they could safely assume that there were not any nasty hidden interactions.

In sum, then, the one behavior common to all our expert subjects that we observed was the experts' repeated use of expectations: they constantly were making conjectures about what they *should* see in the program, based on what is *normally* in an interactive database programs. We have previously argued that knowledge about "what is normal" is represented in terms of programming plans and rules of programming discourse [1]. The expectations, therefore, were derived from these types of knowledge. Given that the experts thus demonstrated their knowledge of what would count as standard programming practices, the reader may be quite surprised at the code patch that was actually produced by these experts.

4. Analyzing the Actual Code Changes

Before turning to an analysis of how our subjects modified the PDB program, let us first analyze the unmodified program. The key issue is the decomposition of the modules: i.e., what is the calling structure of the modules, and why is that an appropriate decomposition? In Figure 4-1 we present a portion of the hierarchy chart (given as part of the documentation) that represents the actual calling sequence of the routines. The chain of routines, GETNME - SRCH - SRCH2, which retrieves a record from the database array, reports back to the main routine; the main routine in turn passes the record to the particular operation, e.g., SHOW. The standard programming principle that was used to structure the code in this way can be phrased as:

```
General Principle: Systematic Grouping Of Functions

Specific Application:
        Code which is independent of each user command (e.g., SHOW,
        UPDATE), should be factored out and  attached to the
        routine that calls the individual command routines.
```

Thus, since the functions of GETNME, SRCH, and SRCH2, routines to get a record name from the user and then find it in database, are used by all the command routines and *contain no reference to any of the specific commands themselves,* they hang off of the main calling routine, and are not called by each of the command routines.

With the above analysis in mind, consider the hierarchy chart and code fragment in Figure

C - 3

4-2 that was generated by 2 of our subjects (2 of the 3 that actually completed the assignment). In contrast to the original situation where the search routines (SRCH and SRCH2) needed to simply fetch an active record (or no record), these routines need to be modified to search for an ACTIVE or DELETED record, *depending on the specific user command*. The patch generated by these subjects was to simply pass the name of the command (in CMD) down through GETNME and SRCH to SRCH2. SRCH2 then tailors its search to the particular command, e.g., if the command is RESTORE then it looks for a DELETED record; if the command is not a RESTORE, then it looks for an ACTIVE record. The retrieval of a record, then, is a function of the the command being acted upon.

The problem of with this method is that it violates the general principle that organized the original modules:

```
General Principle: Systematic Grouping Of Functions

General Principle: VIOLATED!!
   Command is passed down from MAIN to SRCH2;
   command specific information is located in SRCH2
   as well as in the command routines.
```

In other words, information about the command has been distributed outside of the specific command module. Moreover, the structure of the code does not reflect this new functionality: it still appears as if GETNME, SRCH and SRCH2 are independent of the specific commands. Such code structuring can only cause a program reader considerable confusion: since the code appears to be structured according to the modularity principle described above, then one would not expect to find command specific information in routines that were supposed to be command independent. In fact, a program reader using expectations to understand this modified program might easily miss the fact that SRCH2 contains command specific information. On the other hand, an experienced programmer who does notice the distribution of information to SRCH2 might then become quite skeptical of the rest of the program: if a programmer could do *that*, then what else might he do? Finally, given that the programmers exhibited their knowledge of good programming practices in coming to understand the PDB program, one would be quite surprised if, given the task of writing the PDB program from scratch that included the RESTORE command, they would have constructed a program that included their style of patch. Almost certainly a good programmer would have constructed the program using a code structure that is indicated in the patch style given below.

A better coding technique would be to *obey the original structuring principle, and restructure the calling hierarchy*, as is done in Figure 4-3. Since the retrieval of a record is based on the command, therefore the retrieval should be subordinate to the command thus clearly indicating the functional relationship.

```
General Principle: Systematic Grouping Of Functions

Specific Application:
     Since the search routines now need to know about specific commands,
     these search routines should be called by the commands themselves.
```

Each command module now passes a flag to SRCH2 to tell it to search for an ACTIVE or DELETED record. Unfortunately, the little change in the search routines requires a major change to the calling hierarchy. However, the resultant program would be more in keeping with accepted good programming practice and would facilitate the generation of appropriate expectations.

We can summarize the behavior of our subjects with respect to the style of their patch as follows:

> Broadly speaking, there were two constraints on programmers making the enhancement.
>
> - First, there was the code structuring principle that was only *implicit* in the code and documentation.
> - Second, there was a calling hierarchy embodied *explicitly* in the hierarchy chart included in the documentation.
>
> Apparently, our subjects tried to remain consistent to the explicit criteria of the calling sequence reflected in the hierarchy chart, rather than be consistent with the implicit constraint of the code structuring principle.

The obvious implications of this claim are described below.

5. Implications for Documentation

In order to facilitate what we have described as a *model-directed* style of program comprehension, we would suggest that documentation should explicitly contain references to the goals and programming plans in a program, and to the rationale for the choice of those goals and plans. For instance, in the fragment of the PDB program given in the hierarchy chart in Figure 4-1, we must be told something like the following:

```
GOAL: retrieve named record
PLAN: standard item search loop plan
        data structure: array containing database
SUBROUTINE: SRCH2

GOAL: .....
PLAN: .....
SUBROUTINE:  .....

GOAL/PLAN STRUCTURING POLICY:
 General Principle: Systematic Grouping Of Functions

 Specific Application:
       Code which is independent of each user command (e.g., SHOW,
       UPDATE), should be factored out and  attached to the
```

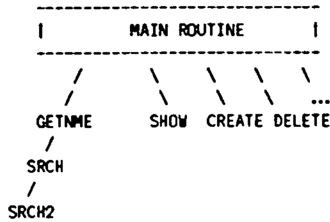
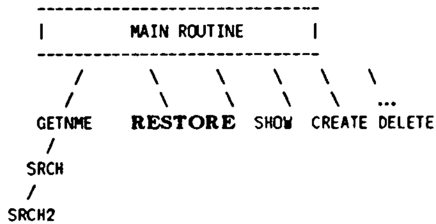
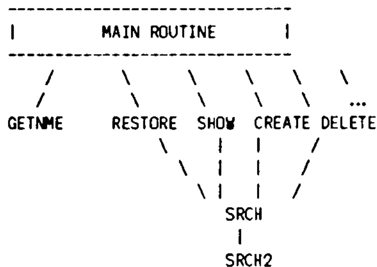


Figure 4-1: Part of Hierarchy Chart of Original Program




```
SUBROUTINE srch2(dbase, ifinal, iptr, name, cmd)
...
DO 700 i=1, ifinal
 IF (name(1 ipos-1) EQ dbase(i 1)(1 ipos-1)
    AND ( (cmd .NEQ. 'r' .AND. dbase(i,7).EQ.'active')
     .OR. (cmd .EQ. 'r' .AND. dbase(i,7) .EQ. 'deleted')
       THEN
         iptr = i
...
END
NOTE  the bold typeface indicates additions to the code made by the maintainers
```

Figure 4-2: Subjects' Modification




```
SUBROUTINE srch2(dbase, ifinal  name  flag  iptr)
...
DO 700 i=1, ifinal
  IF (name(1 ipos-1) EQ dbase(i 1)(1 ipos-1)
     AND dbase(i,7) EQ flag) THEN
        iptr = i
...
END
```

Figure 4-3: A Better Modification

```
routine that calls the individual command routines.
```

This type of documentation makes explicit what the experts are doing anyways: they are searching in the code to verify if the goals they expected are implemented in the manner they expected. Such documentation should enable the program reader to better understand the program.

While that claim may be mildly contentious, consider the following claim: *documentation that contains the goal/plans and their rationale should facilitate better code patches too!* That is, the maintainers will have in front of them explicit reasons why the code is structured in the way it is. Thus, their goal will be to preserve the structuring principles, or at least, be quite clear that they are modifying or violating those principles. When documentation doesn't include that rationale, as we saw in the previous section, subjects were trying to preserve the *surface* results of those deep structuring principles.

6. Concluding Remarks

In this paper we have presented an analysis of the behavior of several expert software maintainers. We have described that behavior in terms of strategies, e.g., model-directed, systematic, and as-needed, that the experts employed in coming to understand the program they were given to modify. We have also presented an analysis of the actual code modification produced by our subjects. The link between these two descriptions can be summarized as follows:

> In making the actual program modification, the maintainers violated their own *good* principles of software construction, which was unfortunately only implicit in the documentation, and instead remained consistent with a structuring constraint (the hierarchy chart) that was explicit in the documentation.

Based on this link, we have suggested how documentation should be changed so as to facilitate the generation of better code patches. We look forward to reporting on subsequent experiments in which we attempt to evaluate the implications of the claims drawn from this first experiment.

ACKNOWLEDGEMENT

The authors would like thank Ed Ng, who smoothed out the logistical bumps we encountered as we conducted this research. Finally, we would like to thank JPL for providing us with the resources to pursue this research.

References

[1] Soloway, E., Ehrlich, K.
Empirical Studies of Programming Knowledge.
IEEE Transactions on Software Engineering SE-10(5):595-609, 1984.

[2] Soloway, E.
A Cognitively-Based Methodology for Designing Languages/Environments/Methodologies.
In *Proc. of then Symposium on Practical Software Development Environments.* ACM SIGSOFT/SIGPLAN, Pittsburgh, Pa., 1984.

PANEL #3

EXPERIMENTS WITH SOFTWARE DEVELOPMENT

K. Koerner, Computer Sciences Corporation
J. Gaffney and S. Martello, IBM Corporation
D. Kafura, Virginia Polytechnical Institute

N86-19975
D8

An Evaluation of Programmer/Analyst Workstations

K. Koerner, R. Mital, and D. Card
Computer Sciences Corporation

A. Maione
National Aeronautics and Space Administration

Computer Sciences Corporation (CSC) and the National Aeronautics and Space Administration (NASA) are striving for improvements in the quality and productivity of software development efforts. Until recently, very few automated tools were available to support software requirements analysis and design even though improvements in quality during these phases appear to offer the greatest leverage for improving the quality and productivity of the overall software development process (Reference 1). Recently, however, some such tools have appeared on the market. This paper documents an effort to evaluate the effectiveness of these tools, specifically programmer/analyst workstations.

As a first step, CSC and NASA studied commerically available products through an industry survey. Next, an in-house evaluation of two commercial products by programmers and analysts was undertaken to determine which tool is the best to support programmers and analysts through life cycle development. Finally, a tool was selected for full implementation on a CSC project, where complete analysis of software statistics over the system life cycle will determine whether or not quality and productivity improvements have actually occurred. This paper summarizes the results of the industry survey and in-house evaluation. Reference 2 describes this study fully.

OBJECTIVES

CSC has adopted a structured software development methodology, summarized in Digital System Development Methodology (DSDM[1]) Reference 3. Part of CSC's commitment to DSDM involves the providing of programmer/analyst workstations that allow this methodology to be implemented easily, thereby permitting programmers and anlaysts to concentrate on technical solutions to problems.

Automated tools can replace the current mode of developing paper models for data flow diagrams, data dictionaries, function specifications, structure charts, and so on. To support the interactive process of analysis and design, the workstations must be able to supply information graphically as well as in text form. Given the iterative nature of anlaysis and design, automation and simplification of the process of generating and refining paper models should increase efficiency. Workstations are the first step in implementing the software factory concept (Reference 4).

To best support DSDM during software development, the analysis and design tools need to automate the basic steps of this methodology. The automated tools ultimately sought should be able to:

- Implement the DeMarco (Reference 5) structured analysis methodology, providing the programmer/analyst with the capabilities to interactively
  - Create and modify data flow diagrams
  - Create and maintain an analysis data dictionary for data flow diagrams.
  - Create and modify process descriptions.
- Implement the Yourdon (Reference 6) structured design methodology, providing the capabities to interactively
  - Create and modify structure charts

[1]DSDM is a trademark of Computer Sciences Corporation.

- Create and maintain a structured design data dictionary.
- Describe a module's design, including a standard format for a prolog in text and a process flow in program design language (PDL).
- Construct a template for a unit test matrix based on the module design.

- Provide these facilities on a microcomputer based workstation -- A basic concept for the programmer/analyst workstation is to be able to implement the tools and techniques of DSDM on a microcomputer workstation. The microcomputer provides the capabilities to
  - Maintain a constant development environment regardless of the project's host computer.
  - Make the tool available to different projects without adding the cost of conversion and retraining.
  - Ensure access at all times -- The project host computer avaiability is eliminated as an issue.
  - Maintain information in a standard format from one project to another -- A project's design is thus maintained on a data base and can be accessed for use on another project.
  - An effective programmer analyst workstation should reduce the cost and improve the quality of requirements analysis and system design activities. Consequently, the overall productivity and reliability of the operational system will increase.

INDUSTRY SURVEY

The industry survey during March-May of 1984, consisted of a two-level screening of commercially available products. This survey phase began with attending conferences, reviewing current literature on the subject, and consulting with technical experts in order to identify feasible resources. Next, sources were screened via telephone discussions and written correspondence. During this initial screening, CSC found that most commercially available products support code generation and report writing. Products or tools that support the development of analysis and design products are fairly new. Many companies indicated that they are pursuing development of these tools on a microcomputer; however, relatively few products are available and supported today. Initially, eight vendors were contacted whose products are currently available in this area. These eight products and their current status as analysis and design tools are listed below.

- Yourdon — -Not available
-Being developed for IBM PC
-Earliest demonstration in January 1985
- Tektronix — -Available for BETA test site
-LSI or VAX based
- PROMOD (GEI) — -U.S. Availability unknown
-IBM PC/XT or VAX based
- Excelerator (Index Technology) — -Available for IBM PC/XT
- CASE 2000 (NASTEC) — -Available on CTEC 8086
- Boeing Argus — -Package and nonsupported source available

	-New enhanced and supported product available in January 1985
• Symbolics	-Available on Symbolics 3600 -No requirements analysis tools
• SOFTOOL CCC and PE	-Configuration control and programming environment tools -IBM PC implementation in late 1984 -Design environment tools in 1985

The initial screening determined that four products met the key criteria of providing requirements analysis and design tools and microcomputer implementation. These were the Tektronix, PROMOD, Excelerator, and CASE 2000.

The second level of the industry survey was to determine which products that met the basic criteria provided the most benefits. CSC had already decided that only an in-house evaluation could provide a sufficiently thorough analysis of benefits. However, further information was needed to determine which products provide sufficient improvements over the current manual approach to warrant the costs associated with an in-house evaluation. Vendor demonstrations were used to determine the availability of the nine major desired feasures at this level of the evaluation. Table 1 shows the desired features and CSC's evaluation of the availablility of each feature for each product.

TABLE 1 - RESULTS OF INDUSTRY SURVEY

FEATURES	TEKTRONIX	PROMOD	EXCELERATOR	CASE 2000
1. *USER FRIENDLINESS*	◒	◒	●	○
2. GRAPHIC AND TEXTUAL DATA MANIPULATION	●	●	●	●
3. CAPABLE INTERACTIVE REQUIREMENTS ANALYSIS TOOLS	●	●	●	●
4. CAPABLE INTERACTIVE DESIGN TOOLS	○	○	●	◒
5. USABILITY AS A DEVELOPMENT TERMINAL ON HOST	●	●	●	●
6. LIBRARY CAPABILITY TO SUPPORT SOFTWARE REQUIREMENTS AND DESIGN TOOLS	◒	◒	●	●
7. MANAGEMENT SUPPORT TOOLS	○	◒	◒	●
8. WORKSTATION NETWORKING CAPABILITIES	●	○	○	●
9. MICROPROCESSOR IMPLEMENTATION	●	●	●	●

○ FEATURE NOT CURRENTLY AVAILABLE

◒ FEATURE PARTIALLY AVAILABLE

● FEATURE AVAILABLE

687-MIT-(59a*)

IN-HOUSE EVALUATION

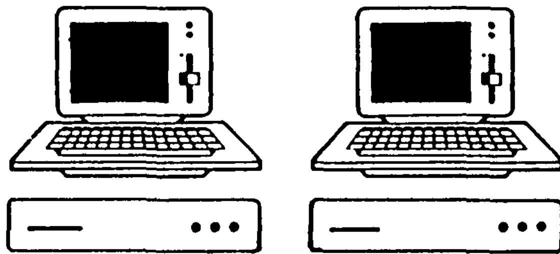
Based on the results of the industry survey the Index Technology Excelerator (Reference 7) and the NASTEC CASE 2000 (Reference 8) workstations were selected for the in-house evaluation. Figure 1 shows the logical configurations of the two systems as implemented for this evaluation. The Excelerator system consisted of two independent workstations (IBM PC/XTs), with individual data bases, and controlled through a mouse interface. The CASE 2000 system consisted of three workstations (CTEC 8086s), connected to a central data base, and controlled through a set of programmed function keys.

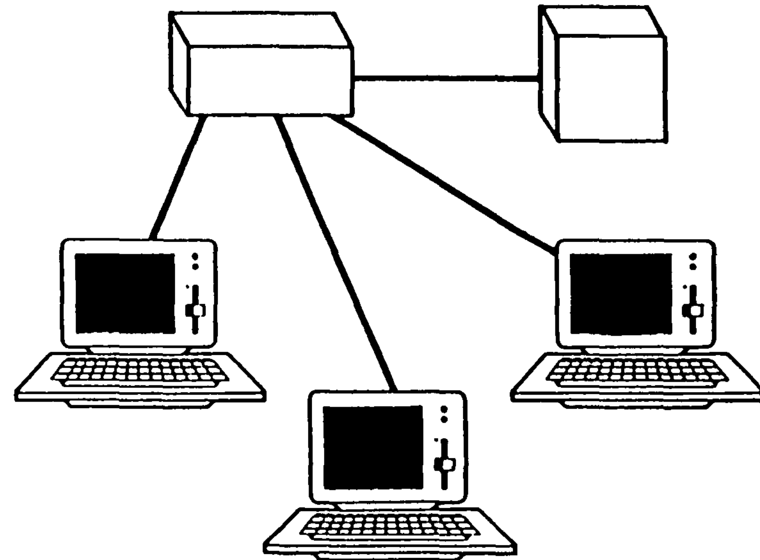
The in-house evaluation included two parts: a general survey of workstation users and a detailed evaluation by a team of experts. Both parts were completed within a three month trial period.

FIGURE 1. CONFIGURATION

EXCELERATOR

CASE 2000





IBM PC/XT

INDEPENDENT WORKSTATIONS

INDIVIDUAL DATA BASES

MOUSE INTERFACE

CTEC 8086

UP TO 16 WORKSTATIONS

CENTRAL DATA BASE

PROGRAMMED FUNCTION KEYS

712-MIT-(89)

User Evaluation

During the three month trial period, the Index Technology Excelerator and NASTEC CASE 2000 workstations were made available to personnel from five different operations. The evaluation organizers did not assign specific problems or times for workstations use. Participants in the evaluation effort generally attempted to apply the workstations to an ongoing task. Relatively few users of either the Excelerator or the CASE 2000 achieved more than 20 hours of contact time.

Users provided their reactions via a questionnaire (reproduced in Reference 2). The questions on this form deal with user background, specific workstation capabilities, overall effectiveness, and the manner in which workstations were used. A total of 34 persons responded to the survey: 22 rated the Excelerator: 29 rated the CASE 2000. Survey respondents represented a wide range of professional experience (from 1 to 20 years). However, most were programmers and/or analysts. Consequently, the requirements analysis and system design capabilities were most carefully explored in this phase of the evaluation.

Survey respondents rated 13 specific tool capabilities as well as the overall effectiveness of each workstation. Table 2 summarizes the respondents' evaluations of the specific tool capabilities. Respondents rated each capability on a scale from one (poor) to five (excellent). Chi-square tests (Reference 9) determined whether or not significant differences ($P<.05$) existed between the workstations with respect to the ratings of each capability.

The Excelerator was rated significantly higher for ease of learning and user friendliness. No substantial differences exist between the two workstations with respect to ratings of requirements analysis and design capabilities. The differences in ease of learning and user friendliness account for the difference in the total ratings shown in Table 2.

TABLE 2 - RESULTS OF USER SURVEYS

CAPABILITY	MEDIAN RATING[a]	
	EXCELERATOR	CASE 2000
GRAPHICS SUPPORT	4	4
EASY TO LEARN	4[b]	2
FAST RESPONSE	3	4
DSDM REQ. ANALYSIS	3	3
DATA FLOW DIAGRAMS	3	3
DSDM DESIGN	3	3
STRUCTURE CHARTS	3	4
DATA DICTIONARY	4	3
USER FRIENDLINESS	4[b]	2
PROJECT MANAGEMENT	—	3[c]
QUALITY ASSURANCE	3	3
CHECK REQUIREMENTS	4	3
CHECK DESIGN	3	3
TOTAL RATING	41	37
NUMBER OF EVALUATORS	22	29

[a]RATING: 5 = GOOD, 1 = POOR.

[b]PROBABILITY < 0.05 THAT THIS DIFFERENCE IN RATINGS IS DUE TO CHANCE.

[c]VALUE NOT INCLUDED IN TOTAL RATING BECAUSE CAPABILITY WAS NOT RATED FOR BOTH WORKSTATIONS.

712-MIT-(66*a)

CSC COMPUTER SCIENCES CORPORATION
SYSTEM SCIENCES DIVISION

Quality assurance and project management capabilities were not fully explored by survey respondents. Users frequently complained of the lack of capabilities for verifying the consistency of requirements and design. Consequently, most survey respondents did not rate these capabilities.

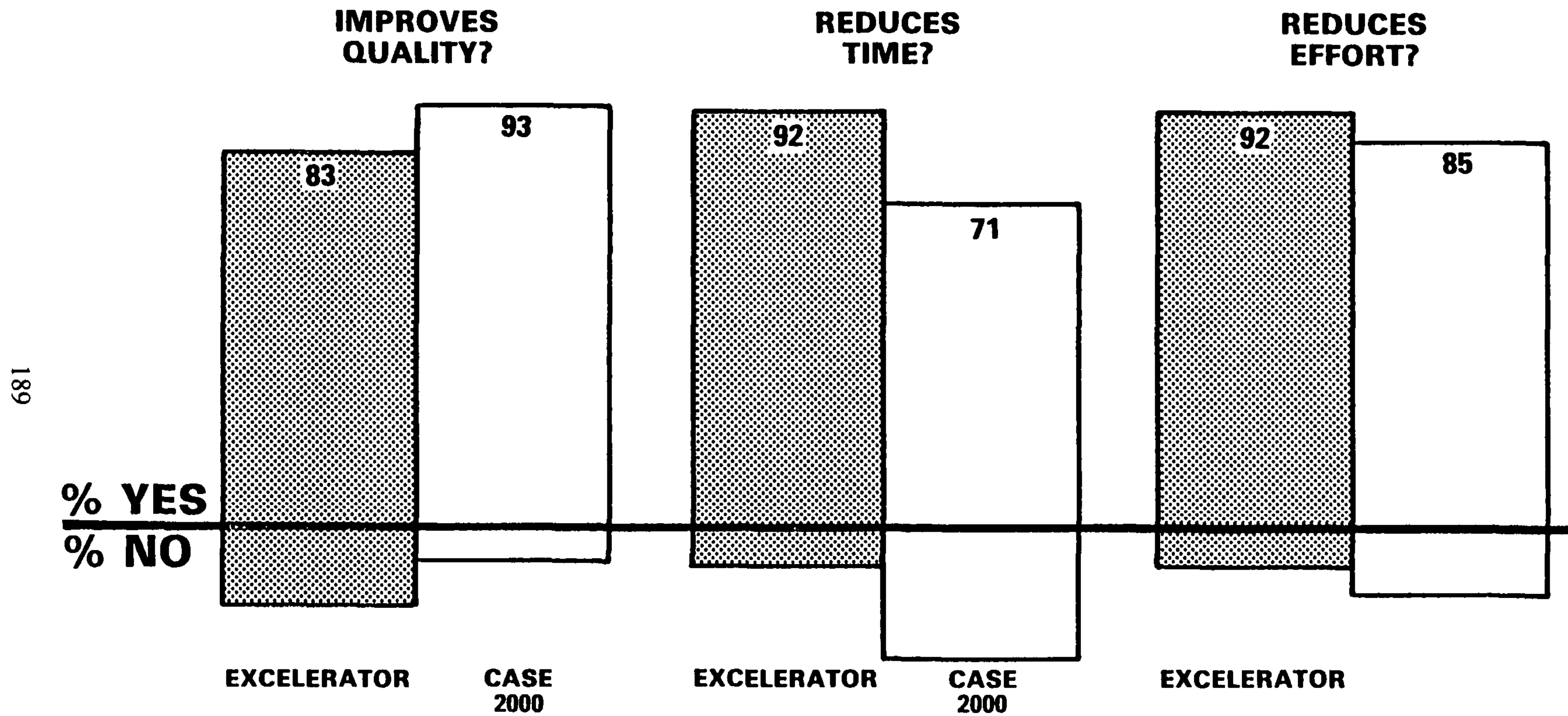
Survey respondents also evaluated the overall effectiveness of the workstations with respect to three key attributes: quality of product, time to generate, and effort to produce. Figure 2 summarizes the responses obtained. These ratings are subjective assessments, not objective measures of actual quality, time and effort. Tests of proportions (Reference 9) determined whether or not the percent of favorable responses was significant ($P < .05$).

Both workstations were judged to be improvements over existing manual procedures, as shown in Figure 2. A significant proportion of respondents rated the Excelerator positively with respect to all three key attributes in spite of frequent complaints about the printer. The CASE 2000 received significant positive ratings for quality and effort only. The lower rating for total time may have been due to the substantial learning time required for operation of the CASE 2000.

In summary, although the Excelerator was rated significantly higher in terms of ease of learning and user friendliness, the two systems were not rated very differently in terms of support for requirements analysis and design. Both systems appeared to offer improvements with respect to the key attributes of quality, time, and effort. However, those individuals who exercised both systems generally stated a preference for the Excelerator.

FIGURE 2 -

USER EVALUATION OF OVERALL EFFECTIVENESS



712-MIT-(89)

CSC COMPUTER SCIENCES CORPORATION
SYSTEM SCIENCES DIVISION

Detailed Evaluation

A detailed comparison of features available on the Excelerator and the CASE 2000 to support requirements analysis and design was undertaken to determine the relative strengths and weaknesses of the two systems. The following paragraphs summarize the approach used, results obtained and conclusions derived from this exercise.

Eight major categories of relevant features were identified:

- Data flow diagrams
- Structure charts
- Data dictionary
- Function specifications
- Data flow diagram validation
- Structure chart validation
- Report/display generation
- General/other

The eight categories were assigned relative weights adding up to 100. Each major category was further divided into specific features. Each feature was assigned a weight of either 1 (desirable) or 2 (mandatroy). Four groups of senior programmers and analysts who had used both the Excelerator and the CASE 2000 fairly extensively during the evaluation period were asked to assess the two systems feature by feature. The input was in the form of both a qualitative assessment as well as a numerical score on a scale of 0 to 5 (0 = not available, 1 = low, 5 = high) for each feature. An informal Delphi method was used to arrive at the ratings on which the results are based.

A final score for each workstation was computed as follows:

Let w_i = weight of ith feature in a major category (value = 1 or 2)

r_i = raw score for ith feature (range = 0 to 5)

W_j = weight of jth major category (W_j = 100)

R_j = overall raw score of jth major category $\left(= \frac{\sum w_i r_i}{\sum w_i}\right)$

Then, final score = $\frac{\sum W_j R_j}{\sum W_j}$

The range of final scores is 0 to 5.

Table 3 shows the computation of final scores from overall raw scores. In summary, the Excelerator and the CASE 2000 scored as follows:

Excelerator	2.01
CASE 2000	2.82

A listing of specific features within the eight major categories and the computation of the overall raw scores for each can be found in Reference 2. One of the major differences between the two systems was the provision for a multi-user centralized data base on the CASE 2000. The evaluation team made two general observations about the workstations:

- Neither the Excelerator nor the CASE 2000 scored very high. This indicates that both systems lack many of the desired features.
- Feature for feature, the CASE 2000 provides more support than the Excelerator.

TABLE 3 - RESULTS OF DETAILED EVALUATION

MAJOR CATEGORY	WEIGHT	OVERALL RAW SCORE		OVERALL WEIGHTED SCORE	
		EXCELERATOR	CASE 2000	EXCELERATOR	CASE 2000
DATA FLOW DIAGRAMS	15	2.5	2.8	37.5	42.0
STRUCTURE CHARTS	15	1.7	3.6	25.5	54.0
DATA DICTIONARY	15	2.5	2.1	37.5	31.5
FUNCTION SPECIFICA-TIONS	5	3.0	5.0	15.0	25.0
DATA FLOW DIAGRAM VALIDATION	15	1.9	2.7	28.5	40.4
STRUCTURE CHART VALIDATION	5	0.0	0.7	0.0	3.5
REPORT/DISPLAY GENERATION	15	1.9	3.1	28.5	46.5
GENERAL/OTHER	15	1.9	2.6	28.5	39.0
TOTAL	100			201	282
FINAL SCORE				2.01	2.82

712-MIT-(59c*)

CONCLUSIONS

The ease and benefit of integrating either workstation into an existing requirements/design environment depend on its match to that environment. The evaluation experience indicated that the Excelerator and CASE 2000 are optimized for different environments. The former targets the environment in which many unrelated, small-to medium-scale requirements/design problems are being solved simultaneously. The latter targets the environment in which the solution to a single large requirements/ design problem is developed over a relatively long period of time.

The Excelerator's ease of learning and operation (via a mouse) makes the system cost effective in those situations in which one or two individuals spend a few months producing a formal requirements/design specification (possibly based on input from a larger team). These individuals spend the rest of their time on other activities (e.g., mathematical analysis or programming). The provision for individual diskettes allows the system to be shared by many users with different problems. Furthermore, the computer can be used to run other software when no requirements/design activity is in progress.

The CASE 2000's central disk and data dictionary support the situation in which many individuals are working on different aspects of the same requirements/design problem. This system simplifies configuration management for large projects and enhances analyst communication. The additional cost imposed by the lengthy training and phase-in period are recovered during the relatively long development period; function keys move the user through the system faster than does a mouse. Furthermore, the function keys can be programmed to satisfy project-specific needs. However, "difficult to learn" implies "easy to forget," so this system is not suited to non-full-time users.

The results of the in-house evaluation indicated that both systems offer improvements in the productivity and quality of requirements analysis and design, relative to the existing manual procedures. These benefits should compound throughout the software life cycle. This is consistent with another recent study (Reference 10) that showed that the availability of workstation support for requirements and design improved overall productivity.

The next step in CSC's evaluation process is to apply these two workstation systems to different production projects of the appropriate sizes. Objective measures of productivity, reliability, and maintainability collected during the development process will enable a quantitative determination of the benefits of workstation usage to be made.

REFERENCES

1. B. W. Boehm, "Software Engineering R&D Trends and Defense Needs." Research Directions in Software Technology. MIT Press: Massachusetts, 1979

2. R. Mital, et al., Programmer/Analyst Workstation Evaluation Report, CSC/TM-84/6138, Computer Sciences Corporation, November 1984

3. Computer Sciences Corporation, Digital System Development Methodology, Version 2.0, March 1984

4. J. H. Manley, "Computer Aided Software Engineering (CASE) Foundation for Software Factories", Proceedings of the Twenty-Ninth International Computer Conference pp 84-91, September 1984

5. T. DeMarco, Structured Analysis and System Specification, Yourdon Press, 1978

6. E. Yourdon and L. Constantine, Structured Design, Yourdon Press 1978

7. Index Technology Corporation, Excelerator Reference Guide, 1984

8. NASTEC Corporation, CASE 2000 Workstation Reference Manual, Release 3.0, July 1984

9. H. M. Blalock, Social Statistics, McGraw-Hill: New York, 1972

10. B. W. Boehm, et al., "A Software Development Environment for Improving Productivity" IEEE Computer, pp 30-42, June 1984

N86-19976

D9

A MODEL FOR THE PREDICTION OF LATENT ERRORS USING DATA OBTAINED DURING THE DEVELOPMENT PROCESS

Presentation At: The 9th Annual Software Engineering Workshop,
NASA, Goddard, Nov. 28, 1984

John E. Gaffney, Jr.
IBM Corporation
Federal Systems Division
Advanced Technology Dep't.
Gaithersburg, Maryland

Steven J. Martello
IBM Corporation
Kingston, New York

SUMMARY

This paper presents a model implemented in a program that runs on the IBM PC for estimating the latent (or post ship) content of a body of software upon its initial release to the user. The model employs the count of errors discovered at one or more of the error discovery processes during development, such as a design inspection, as the input data for a process which provides estimates of the total life-time (injected) error content and of the latent (or post ship) error content--the errors remaining at delivery.

The software development process may be considered to consist of a sequence of activities. One set is that used in the IBM, Federal Systems Division, [1, 2] which is: system definition, software design, software development (coding and unit test), software system test, and system/ acceptance test. Included in these major activities are error discovery processes. A set of them is:

1. High level design inspections
2. Low Level design inspections
3. Code inspections
4. Unit test
5. Integration test
6. System test

The model presented here presumes that these activities cover all of the opportunities during the software development process for error discovery (and removal). Data will, typically, not be available in all of them for any particular project. The model might be expanded to cover some additional software error discovery activities, such as a "requirements/ objectives inspection"; that possibility will not be considered further here, however.

Analysis of the number of errors discovered at the successive stages of the software development process suggest that the profile of defect discovery during the software development process, when taken on a phase-by-phase basis, at first increases and then decreases as a function of phase (e.g., high level design inspection, low-level design inspection, etc.). Thus, errors per KSLOC (thousands of source lines of code) may be plotted as a function of each error discovery phase or activity as shown in Figure 1.

The model employs a discrete form of the Rayleigh curve to represent the errors/KSLOC removed as a function of defect removal process. It is of interest to note that the Rayleigh curve has been used widely to model the "proper" application of manpower to develop processes in general [3] and the software development process more particularly [4] as well as the entire software development life cycle. [5] The model presented here does not presume any given "level" of the SLOC to which it is applied (e.g., JOVIAL vs. assembly "level" code). A recent paper by Gaffney [6] presents an analysis of some software data that suggests that the error content of a body of software is strongly a function of the number of SLOC and not of the "level" of the language in which they are written. The cumulative form of the Rayleigh model, as applied to the defect discovery process model presented here, is:

$$V_t = E * \left(1 - e^{-bt^2}\right)$$

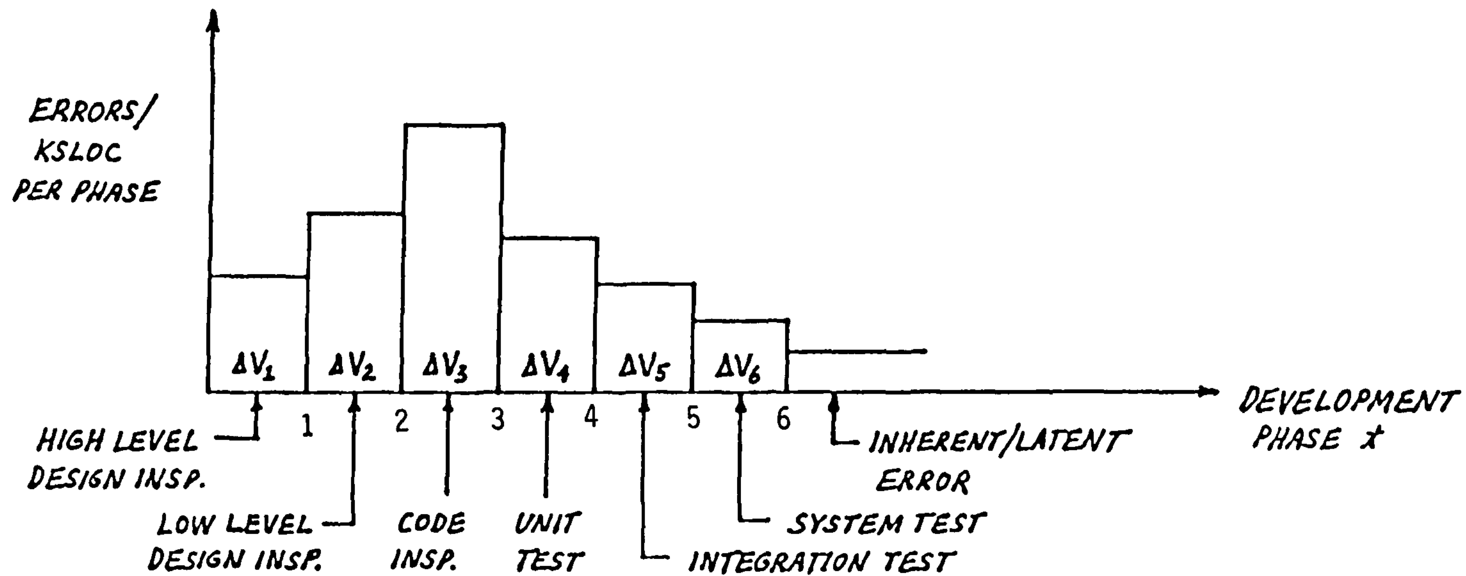
where;

V_t = total number of errors (or errors per KSLOC) discovered through development phase (or activity no. "t").

E = total lifetime defect content or "injected" error.

b = $\frac{1}{2\tau_d^2}$; τ_d = "error discovery phase constant," the point at which 39% of "E" errors has been discovered.

FIGURE 1-ERROR DISCOVERY PROFILE



RAYLEIGH CURVE FIT: $\Delta V_t = E\left[e^{-b(t-1)^2} - e^{-bt^2}\right]$

E = TOTAL LIFETIME ERROR RATE

$b = \frac{1}{2T_d^2}$; T_d = DEFECT DISCOVERY PHASE CONSTANT

The independent variable, "t", represents error discovery activity indices as follows:

t	Error Discovery Activities
1	High level design inspections
2	Low level design inspections
3	Code Inspections
4	Unit Test
5	Integration Test
6	System Test
$6 \rightarrow \infty$	Field Potential or Latent Errors

The Rayleigh curve may be expressed so that it can be used to model discrete data groupings (corresponding to the discrete activities of the software development process) as follows:

Let U_t be the actual number of errors discovered, defects per KSLOC noted, PTR's per KSLOC written, or other convenient measures of defect removal during phase t (which extends from "time" or "activity index value" (t-1) to t). The "idealized" equivalent to this value, given by the discrete Rayleigh model is ΔV_t, where:

$$\Delta V_t = E*(e^{-b(t-1)^2} - e^{-bt^2})$$

The idea of the model is to estimate "b" and E from data obtained during one or more of the error discovery processes listed above, and then use the equation for ΔV_t to estimate the error discovery rates (errors/KSLOC) for the remaining error discovery processes.

The software error discovery profile model presented here can be used to aid in the management and control of the software development process by providing projections of the number of errors that will be found at later stages of the development process, based upon discovery data from earlier stages. If the error discovery rates are not as high as earlier

projected, this may suggest that some management action is appropriate, such as scheduling additional inspections, extending the number of test hours, etc.

The approach taken in the model offers a number of advantages relative to various possible alternatives to the software developer in gaining an understanding of the error creation and removal processes associated with his software product. The model would facilitate an early estimate of error content; the developer need not wait until the software is actually coded and is running in a processor. He can use data obtained during inspections to gain knowledge about the probable error content of his software upon its release. If he is not pleased, he has time to take actions that will, hopefully, counter that situation. Data about different segments of a software product can be combined, and/or compared, as appropriate, since a time base is not directly involved. This feature of the model also facilitates the comparison of different software products' error discovery histories to be made more easily than might otherwise be possible if the error data were time-based oriented.

The excellent work of Mr. Rick Qualters of IBM,Gaithersburg in implementing the model to run on the IBM PC is gratefully acknowledged.

REFERENCES

1. Quinnan, R.E., "The Management of Software Engineering, Part V," "IBM Systems Journal," Vol. 19, No. 4, 1980; pg. 466.

2. Gaffney, J. E., Jr., "Approaches to Estimating And Controlling Software Costs"; CMG XIV Proceedings, op cit, pg. 335.

3. Norden, P.V., "Useful Tools for Project Management," in M. K. Storr, Ed., "Management of Production," Penguin Books, New York, 1970.

4. Gaffney, J. E., Jr., "A Macroanalysis Methodology for Assessment of Software Development Costs," in "The Economics of Information Processing," Volume 2, pg. 177, John Wiley and Sons, New York, 1982, edited by R. Goldberg and H. Lorin.

5. Putnam,L.H. , "A General Empirical Solution to the Macro Software Sizing and Estimating Problem," "IEEE Transactions on Software Engineering"; SE-4 (4); July, 1978, pg. 345.

6. Gaffney, J. E., Jr., "Estimating the Number of Faults in Code," "IEEE Transactions on Software Engineering," Vol. SE-10, No. 4, July 1984, pg. 459.

THE VIEWGRAPH MATERIALS

for the

J. GAFFNEY/S. MARTELLO PRESENTATION FOLLOW

A MODEL FOR THE PREDICTION OF LATENT ERRORS USING DATA OBTAINED DURING THE DEVELOPMENT PROCESS

Presentation At: The 9th Annual Software Engineering Workshop,
NASA, Goddard, Nov. 28, 1984

John E. Gaffney, Jr.
IBM Corporation
Federal Systems Division
Advanced Technology Dep't.
Gaithersburg, Maryland

Steven J. Martello
IBM Corporation
Kingston, New York

PLEASE NOTE:

VARIOUS NUMBERS PRESENTED SUBSEQUENTLY SHOULD NOT BE INTERPRETED AS ACTUAL IBM WORKING DATA, BUT ARE PROVIDED FOR ILLUSTRATION ONLY.

THIS TALK PRESENTS:

WHAT: A METHOD/MODEL FOR ESTIMATING:

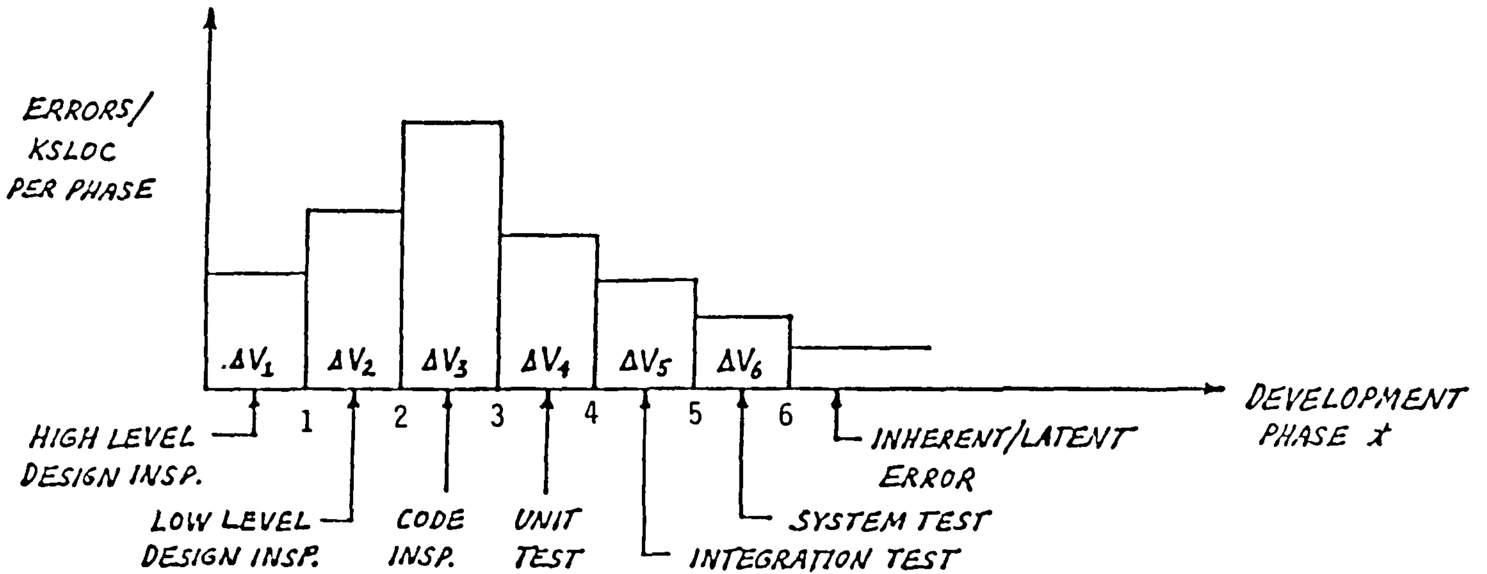
THE LIKELY ERROR CONTENT OF A BODY OF SOFTWARE UPON ITS DELIVERY TO A USER, BASED ON DATA OBTAINED DURING THE DEVELOPMENT CYCLE.

WHY: AN ESTIMATE, EARLY IN THE DEVELOPMENT CYCLE, OF ERROR CONTENT/SYSTEM UNAVAILABILITY CAN BE A VERY IMPORTANT INPUT TO THOSE CONTROLLING THE SYSTEM DEVELOPMENT PROCESS.

SOFTWARE DEVELOPMENT CYCLE

DEVELOPMENT ACTIVITIES	INCLUDED ERROR DISCOVERY/ REMOVAL ACTIVITIES
SYSTEM DEFINITION	-
SOFTWARE DESIGN	HIGH LEVEL DESIGN INSPECTION LOW LEVEL DESIGN INSPECTION
SOFTWARE DEVELOPMENT (CODING & UNIT TEST)	CODE INSPECTION UNIT TEST
SOFTWARE SYSTEM TEST	SOFTWARE INTEGRATION TEST
SYSTEM & ACCEPTANCE TEST	SYSTEM TEST

ERROR DISCOVERY PROFILE



RAYLEIGH CURVE FIT: $\Delta V_t = E\left[e^{-B(t-1)^2} - e^{-Bt^2}\right]$

E = TOTAL LIFETIME ERROR RATE

$B = \frac{1}{2T_d^2}$; T_d = DEFECT DISCOVERY PHASE CONSTANT

RAYLEIGH CURVE, CUMULATIVE FORM:

$V_t = E*(1-e^{-Bt^2})$

LATENT ERROR CONTENT:

L=E*e-36B

RANGE OF DEVELOPMENT PHASE INDEX (t)	ERROR DISCOVERY/REMOVAL ACTIVITY
0-1	HIGH LEVEL DESIGN INSPECTION
1-2	LOW LEVEL DESIGN INSPECTION
2-3	CODE INSPECTION
3-4	UNIT TEST
4-5	INTEGRATION TEST
5-6	SYSTEM TEST
$6 \rightarrow \infty$	LATENT/POST-SHIP ERROR

'BASELINE' ERROR DISCOVERY PROFILE[1]

PERCENT OF LIFETIME ERROR CONTENT	ERROR DISCOVERY/REMOVAL ACTIVITY
7.69	HIGH LEVEL DESIGN INSPECTION
19.70	LOW LEVEL DESIGN INSPECTION
23.93	CODE INSPECTION
20.88	UNIT TEST
14.27	INTEGRATION TEST
7.92	SYSTEM TEST
5.61	LATENT/POST-SHIP ERROR

NOTE (1): FOR $\tau_D = 2.5$; $B = .08$

ESTIMATION OF TOTAL LIFETIME ERROR RATE, E

- o BOTH E & B (THE PEAK LOCATION CONSTANT) ARE ESTIMATED BY OBTAINING A 'BEST FIT' TO THE DATA, THE USER-ENTERED VALUES,

  E.G.: US_1 = HIGH LEVEL DES. INSPEC. ERRORS/KSLOC.

  BEST FIT <=> E & B SUCH THAT

$$D = \sum_I D_I^2 \;; \quad D_I = (US_I - \Delta V_I)$$

  D = MINIMUM

- o THEN, FOR EXAMPLE:

  IF HIGH LEVEL DES. INSPECTION AND CODE INSPECTION DATA ARE AVAILABLE:

$$\hat{E} = \frac{US_1 + US_3}{(1 - e^{-B} + e^{-4B} - e^{-9B})}$$

<u>USING THE VALUE B = .08 FOR THE BASELINE ERROR DISCOVERY PROFILE, WE WOULD HAVE</u>:

$$\hat{E} = \frac{US_1 + US_3}{(1 - e^{-.08} + e^{-.32} - e^{-.72})}$$

SAMPLE ESTIMATE USING PUBLISHED DATA

- o GAFFNEY[1] (USING LIPOW[2] DATA) SUGGESTED 22.7 ERRORS PER KSLOC AFTER CODE COMPILATION

IMPLIES: $\hat{E} = \frac{22.7}{.4868} = 46.6$ ERRORS/KSLOC

$\hat{L} = 46.6 \times .0561 = 2.62$ ERRORS/KSLOC

- o SCHNEIDER[3] SUGGESTED A FIGURE OF 20 ERRORS/KSLOC COMMENCING WITH UNIT TEST

IMPLIES: $\hat{E} = \frac{20.0}{.4868} = 41.1$ ERRORS/KSLOC

$\hat{L} = 41.1 \times .0561 = 2.3$ ERRORS/KSLOC

THESE FIGURES ARE RELATIVELY CLOSE.

NOTES: (1): IEEE SOFTWARE ENG. TRANSACTIONS; JULY, 1984
(2): IEEE SOFTWARE ENG. TRANSACTIONS; JULY, 1982
(3): ACM/SIGMETRICS PER, SPRING, 1981

SOME OVERALL OBSERVATIONS

- DATA ANALYZED BY PRINCIPAL DEVELOPMENT ACTIVITY, RATHER THAN BY TIME AS INDEPENDENT VARIABLE.

- AVOIDS DETERMINING 'EQUIVALENCE' OF TIME BASES IN INSPECTIONS, SWIT, ETC.

- FACILITATES EARLY ESTIMATE OF DEFECT CONTENT.

- AVOIDS MANAGEMENT PROBLEM OF ASKING DEBUGGERS TO NOTE PRECISE TIMES OF DEFECT DETECTIONS.

- ENABLES (STATISTICAL) ADVANTAGE TO BE TAKEN OF GROUPING DEFECT DETECTIONS - MINIMIZES EFFECT OF 'NOISE' IN DATA.

- ENABLES COMPARISON OF DIFFERENT PROJECTS' ERROR DETECTION HISTORY TO BE MADE WITHOUT REGARD TO SCHEDULE DIFFERENCES.

N86-19977
D10

THE INDEPENDENCE OF SOFTWARE METRICS TAKEN AT DIFFERENT LIFE-CYCLE STAGES

by

Dennis Kafura
James Canning
Gereddy Reddy

Computer Science Department
Virginia Polytechnic Institute
Blacksburg, VA 24061

ABSTRACT

Over the past few years a large number of software metrics have been proposed and, in varying degrees, a number of these metrics have been subjected to empirical validation which demonstrated the utility of the metrics in the software development process. In this paper we will report on our attempts to classify these metrics and to determine if the metrics in these different classes appear to be measuring distinct attributes of the software product. Statistical analysis is used to determine the degree of relationship among the metrics.

I. INTRODUCTION

Underlying our effort to determine an operational classification of the metrics is the belief that software products exhibit different forms of "complexity" in a number of "independent" dimensions. That there are a number of different forms of "complexity" is attested to by the large number of different complexity metrics and also by a number of prior studies of programmer performance. That these complexities are "independent" is evidenced by the common practice of trading one form of complexity for another in the design and implementation processes. For example, the global relationships between components can often be simplified by combining several together. However, this simplification of the global relationships results in greater complexity within the newly created component.

A proper classification of software metrics is important for two reasons. First, it reduces the number of metrics which must be employed. The costs associated with redundant metrics include not only the price of extracting, storing and displaying the metric but also, and perhaps more importantly, the price to an analyst of viewing and attempting to evaluate the significance of these additional metrics. Second, the elimination of redundant metrics focusses our attention fundamental factors which affect software complexity, leads more directly to the discovery of other independent metrics, and simplifies the processes of investigating and modeling of the software development process and its products.

The study which we report in this paper is also of interest because of it uniquely combines the following features. First, a variety of software metrics are used including metrics of low-level code details as well as measures of general relationships between components. Second, only realistic software systems are used. Realisitic systems to us are those that have been developed by several individuals over more than a year of calendar time in some demanding application area. Third, evidence from different application area and different development environments is presented. The three systems presented in this paper are an operating system, a database system, and ground-support software systems. Fourth, because of the size of the systems considered and the number of metrics evaluated it was important to develop automated metric tools. Fifth, and finally, we have applied the metrics to the same collection of software systems making it possible to compare these metrics.

The metrics which we considered may be group into three broad classes based on the features of the software product which must be known in order to compute the metric. The metrics in one class, referred to as code metrics, are defined in terms of the features of the implemented code. Metrics in this class include Haltead's software science measures [1] and McCabe's cyclomatic complexity measure [2]. A second class of metrics, termed structure metrics, is based on more global features of the software system. Typically,

structure metrics are defined in terms of some relationship between major components of the system without regard for the details of the components themselves. This class includes Henry and Kafura's information flow complexity measure [3-5] and McClure's invocation complexity measure [6]. Hybrid metrics, the third major class, combines elements of both code and structure metrics. A member of this class, Woodfield's syntactic interconnection measure [7], combines control and data relationships between components with Halstead's effort measure to produce a composite measure for each component. Yau and Collofello's stability measure [8] is also in this class.

II. COMPARISON OF METRICS

The results of this study are briefly: (1) the code metrics studied all appear to be high associated, and (2) the structure and hybrid metrics appear to be distinct among themselves and different from the code metrics.

The first of three sets of data is shown in Table 1. This data is based on an analysis of the kernel of the UNIX operating system and has appeared previously [9]. This table shows a comparison between only one of the structure metrics, the information flow complexity, and a variety of code metrics. The code metrics used in this study included several of the Halstead software science measures and McCabe's cyclomatic complexity. The last column in this table shows that a low correlation exists between the information flow metric and any of the code metrics. The range of correlations is 0.20 to 0.38. On the other hand, very strong relationships appear among the code metrics themselves. The correlations among the code metrics ranges from 0.84 to 0.99. This study was the first evidence that a classification of software metrics was both possible and necessary.

The second set of data is presented in Tables 2 and 3. The metrics used in this study were derived from an automated analysis of a database management system constructed at Virginia Tech [10]. This system has undergone 4 major revisions over a period of approximately five years. The code metrics used in these tables include only one of the Halstead measures, the effort measure, along with the length (lines of code) and McCabe's cyclomatic complexity. In contrast to the first study, however, all of the structure and hybrid metrics have been included in this experiment. An additional factor in this experiment is the use of two different types of statistical measures, the Pearson parametric measure and the Spearman non-parametric measure. As can be seen by comparing Tables 2 and 3 the essential results are the same regardless of the statistical measure used. It may be observed in this two tables that the code metrics are again highly associated. the Pearson correlations range between 0.79 and 0.97 while the Spearman correlations range between 0.81 and 0.95. Also apparent from these tables is the marked lack of association between the code metrics and the structure or hyrbid metrics. The range of Pearson correlations in

this case is from 0.15 to 0.49 while the Spearman correlations range between 0.05 and 0.28. Also it should be noticed that the associations among the structure and hybrid metrics is weak. Except in the case of the correlation of 0.60 between the information flow metric and McClure's metric, the range of Pearson correlations for these two groups of metrics is from -0.06 to 0.36 while the Spearman correlation lie between -0.05 and 0.43.

The study of the database management system strengthen our conviction that a clear distinction exists between measures based on code details and measures based on more global relationships among components. Furthermore, this study also leads one to believe that the measures of global relationships are measuring different properties of the software system. Confirmation of these results is sought in the last of the three studies presented in this paper.

The final set of data was derived from a study of several ground support software systems developed by the Computer Sciences Corporation for NASA Goddard in cooperation with the Software Engineering Laboratory. A typical set of data from this extensive study is shown in Table 4.

An examination of Table 4 shows that, once again, a strong association exists among the code metrics. Somewhat in contrast to the previous data, however, we observe a higher level of association between the code metrics and the information flow metric. This one aside, the range of correlations between the code and structure metrics is from 0.16 to 0.51. With regard to the information flow metric is should be observed that even though higher correlation were seen in this study than in the two previous ones, the level of correlations (0.55 to 0.63) is still significantly lower than the correlations among the code metrics (0.85 to 0.96). Furthermore, if Pearson correlation coefficients are used, the level of correlation between the code metrics and the information flow metric falls into the range of 0.26 to 0.45 – certainly comparable to the previous data. Finally, the relationship between the structure and hybrid metrics has one anomalous point – a 0.71 correlation between the information flow metric and the Yau and Colofello stability measure. Aside from this one point, the range of correlations between these two classes of metrics is from 0.20 to 0.47. .para For the most part, the study of the Goddard systems is consistent with the results seen in the prior two studies. Only one metric, the information flow metric, exhibited a somewhat different pattern than had appear earlier.

III. CONCLUSIONS

Based on the data presented in the paper we feel confident in concluding that: (1) the code metrics considered in this study are measuring essentially the same properties of software systems; and (2) the structure and hybrid metrics considered in this study are measuring properties of the software system distinct from the code metrics and also from each other. These two conclusions are advanced with some confidence since the same results have been observed in software systems which were written in two different languages (C and Fortran), were developed in different time frames for different application areas and in different development environments with different personnel.

Based on these results we would argue that less work needs to be done in inventing new metrics based on code details and that more work must be done to establish a more complete set of "independent" metrics. It is by no means to be implied by our study that the set of structure and hybrid metrics which we have used is in any sense complete.

IV. REFERENCES

[1] Halstead, M.H. *Elements of Software Science*, Elsevier, New York, 1977.

[2] McCabe, T.J. "A Complexity Measure," *IEEE Transactions on Software Engineering*, SE-2, December 1976.

[3] Henry, S.M. and Kafura, D.G. "Software Structure Metrics Based on Information Flow," *IEEE Transactions on Software Engineering*, SE-7, September 1981.

[4] Kafura, D.G. and Henry, S.M. "Software Quality Metrics Based on Interconnectivity," *The Journal of Systems and Software*,, 2, 1981.

[5] Henry, S.M. and Kafura, D.G. "The Evaluation of Software Systems' Structure Using Quantitative Software Metrics," *Software– Practice and Experience*, Vol. 14(6), June 1984.

[6] McClure, C. "A Model for Program Complexity Analysis," *Proceedings: 3rd International Conference on Software Engineering*, Atlanta, Georgia, May 1978.

[7] Woodfield, S.N. "Enhanced Effort Estimation by Extending Basic Programming Models to Include Modularity Factors," *Ph.D. Thesis*, Purdue University, 1980.

[8] Yau, S. and Collofello, J. "Some Stability Measures for Software Maintenance," *IEEE* Transactions on Software Engineering, Vol. SE-6, No. 6, November 1980.

[9] Kafura, D.G., Henry, S.M., and Harris K. "On the Relationship Among Three Software Metrics," *1981 ACM Symposium on Measurement and Evaluation of Software Quality*, University of Maryland, March 25-27, 1981.

[10] Reddy, G.R. "Application of Software Quality Metrics to a Relational Data Base System," *Master's Thesis*, Virginia Polytechnic Institute, May 1984.

	N	N(est.)	Volume	Effort	McCabe	In.Flow
N	1.0	.94	.99	.92	.91	.32
N(est.)		1.0	.94	.81	.84	.20
Volume			1.0	.94	.91	.31
Effort				1.0	.84	.38
McCabe					1.0	.34
In.Flow						1.0

Table 1. UNIX Study (Pearson Correlations)

	Length	Effort	McCabe	In.Flow	Wood.	Yau	McClure
Length	1.0	.97	.79	.49	.26	.17	.49
Effort		1.0	.87	.39	.26	.24	.43
McCabe			1.0	.15	.24	.24	.26
In.Flow				1.0	.19	-.06	.60
Wood.					1.0	.00	.36
Yau						1.0	.07
McClure							1.0

Table 2. DataBase Management System Study (Pearson Correlations)

	Length	Effort	McCabe	In.Flow	Wood.	Yau	McClure
Length	1.0	.95	.81	.26	.04	.33	.26
Effort		1.0	.82	.36	.08	.37	.23
McCabe			1.0	.21	.05	.39	.28
In.Flow				1.0	.39	.43	.14
Wood.					1.0	-.05	.29
Yau						1.0	.11
McClure							1.0

Table 3. DataBase Management System Study (Spearman Correlations)

	Length	Effort	McCabe	In.Flow	Wood.	Yau	McClure
Length	1.0	.96	.86	.62	.20	.49	.46
Effort		1.0	.85	.63	.18	.51	.44
McCabe			1.0	.55	.16	.46	.40
In.Flow				1.0	.38	.71	.46
Wood.					1.0	.26	.20
Yau						1.0	.47
McClure							1.0

Table 4. NASA Goddard Study (Spearman Correlations)

THE VIEWGRAPH MATERIALS

for the

D. KAFURA PRESENTATION FOLLOW

THE INDEPENDENCE OF SOFTWARE METRICS TAKEN AT DIFFERENT LIFE—CYCLE STAGES

Dennis Kafura
James Canning
Gereddy Reddy

Virginia Polytechnic Institute
Blacksburg,Virginia

HYPOTHESES ABOUT SOFTWARE METRICS

- **Software Systems are complex entities with a number of "independent" dimensions of "complexity".**

- **Many kinds of "complexity" have tangible attributes which can be quantified (i.e., measured).**

GOAL

Find a "complete" and "minimal" set of metrics

Complete : all forms of complelxity are measured

Minimal : no redundant metrics

QUESTION

Are metrics currently in use "independent" of each other ?

APPROACH

- **Use automated tools to obtain metrics of realistic software systems**

- **Include a variety of metrics**
  - **code (Halstead,McCabe, etc.)**
  - **structure (information flow, McClure)**
  - **hybrid (Yau, Woodfield)**

- **Study statistical relationship among metrics**

	N	N(est.)	Volume	Effort	McCabe	In.Flow
N	1.0	.94	.99	.92	.91	.32
N(est.)		1.0	.94	.81	.84	.20
Volume			1.0	.94	.91	.31
Effort				1.0	.84	.38
McCabe					1.0	.34
In.Flow						1.0

Table 1. UNIX Study (Pearson Correlations)

	Length	Effort	McCabe	In.Flow	Wood.	Yau	McClure
Length	1.0	.97	.79	.49	.26	.17	.49
Effort		1.0	.87	.39	.26	.24	.43
McCabe			1.0	.15	.24	.24	.26
In.Flow				1.0	.19	-.06	.60
Wood.					1.0	.00	.36
Yau						1.0	.07
McClure							1.0

Table 2. DataBase Management System Study (Pearson Correlations)

	Length	Effort	McCabe	In.Flow	Wood.	Yau	McClure
Length	1.0	.95	.81	.26	.04	.33	.26
Effort		1.0	.82	.36	.08	.37	.23
McCabe			1.0	.21	.05	.39	.28
In.Flow				1.0	.39	.43	.14
Wood.					1.0	-.05	.29
Yau						1.0	.11
McClure							1.0

Table 3. DataBase Management System Study (Spearman Correlations)

	Length	Effort	McCabe	In.Flow	Wood.	Yau	McClure
Length	1.0	.96	.86	.62	.20	.49	.46
Effort		1.0	.85	.63	.18	.51	.44
McCabe			1.0	.55	.16	.46	.40
In.Flow				1.0	.38	.71	.46
Wood.					1.0	.26	.20
Yau						1.0	.47
McClure							1.0

Table 4. NASA Goddard Study (Spearman Correlations)

PANEL #4

SOFTWARE TOOLS

W. Farr, NSWC
L. Putnam, QSM
D. Levine, Intermetrics

N86-19978
D11

AN INTERACTIVE PROGRAM FOR SOFTWARE RELIABILITY MODELING

by

William H. Farr

and

Oliver D. Smith

ABSTRACT

With the tremendous growth in computer software, the demand has arisen for producing cost-effective reliable software. Over the last 10 years an area of research has developed which attempts to address this problem by estimating a program's current reliability by modeling either the times between error detections or the error counts in past testing periods. This paper describes a new tool for interactive software reliability analysis using the computer. This computer program allows the user to perform a complete reliability analysis using any of eight well-known models appearing in the literature. The paper illustrates some of the capabilities of the program by means of an analysis of a set of simulated error data.

CONTENTS

Page

INTRODUCTION 1

SMERFS' GOALS AND DESIGN 3
MAINTAINABILITY 3
COMPLETE RELIABILITY ANALYSIS ENVIRONMENT 4
INTERACTIVE IN NATURE 4
ERROR DETECTION CAPABILITIES 6
MACHINE TRANSPORTABILITY 6

SAMPLE DATA ANALYSIS 6

SUMMARY 18

REFERENCES 19

ILLUSTRATIONS

Figure Page

1 PROGRAM STRUCTURE 5

2 PROGRAM MENU 8

3 DATA INPUT 9

4 SUMMARY STATISTICS 10

5 PLOTS OF RAW DATA 11

6 SELECTING A MODEL 13

7 MODEL ESTIMATION PROCEDURES 14

8 GOODNESS-OF-FIT 15

9 MODEL FIT OF DATA 16

10 PLOT OF RESIDUALS 17

INTRODUCTION

Over the last decade there has been a tremendous growth in the applications of computer software. Part of this growth has been due to the development of microprocessors and distributed processing and networking. Every day new and innovative ideas on how the computer can be applied in business, education, industry, and government are being proposed. This "computer revolution" has spurred the dramatic growth in the number, size, and complexity of the accompanying computer software. In 1977 the costs of just the software to the entire U.S. economy ranged from 10 to 19 billion dollars (Reference 1).

This increasing role for software has also meant the emergence of the problem of developing "error-free" programs. For large, complex programs the number of conceivable logic paths through the code is astronomical, making it impossible to check every path for correctness. Researchers and practitioners of software code development have therefore looked for ways of minimizing the chances of error introduction in the program design and development stages. Various tools and approaches used to accomplish this include: structured code, "top-down" design, and the development of a number of automated verification and validation (V&V) tools for program checkout. Another area of research, which attempts to quantify the degree to which a section of code is "error-free," is software reliability estimation. Software reliability is defined as "the probability that a given software program will operate without failure for a specified time in a specified environment." A software failure is defined as "any occurrence attributable to software in which the system did not meet its performance requirements." If one were to have an idea of a program's current reliability, a more rational judgment could be made on when that software should be released to the user. Moreover, knowing the reliability of the various components of a program could aid the testing team in making determinations for allocations of testing personnel and time to those sections of the code in which the indicated reliability is low.

Over the last 15 years many models and estimation procedures have been proposed to quantify a program's reliability. References 2 through 6 are excellent reviews of the various approaches. The approach that has received the greatest emphasis in the literature centers upon modeling either the times between error detections [measured either by elapsed wall clock time or Central Processing Unit (CPU) time] or the number of errors detected per testing period. In addition to estimates of a program's reliability, these models usually estimate the total number of errors in the code and the expected time (or number of errors) until the next error detection (in the next testing period).

Many of these models are either based upon the assumption that the time between errors follows an Exponential distribution or the number of detected errors per testing period follows a Poisson distribution. The parameters of these distributions are taken as functions of up to three unknowns. The unknowns are estimated using either a maximum likelihood or least squares procedure. The estimates are then used to estimate the reliability measures of the program. A major problem with these models is the difficulty in obtaining the estimates. Many of these models are nonlinear in the unknowns, thus requiring sophisticated numerical techniques to obtain the estimates. This necessitates the use of the computer and thus the primary reason for developing an interactive computer program for software reliability modeling. Different starting points for the numerical procedures can be input allowing the user to investigate the optimality of the achieved estimates. Once the user is satisfied that the appropriate estimates have been obtained, various reliability estimates are provided along with the associated precision of the estimates.

The Statistical Modeling and Estimation of Reliability Functions for Software (SMERFS) program incorporates eight different models; four using as input data the times between error occurrences and four using the number of detected errors per testing period. The former include: Littlewood and Verrall's Model (Reference 7), Moranda's Geometric Model (Reference 8), John Musa's Execution Time Model (Reference 9), and an adaptation of Goel's Non-Homogenous Poisson Process (NHPP) Model to time between error data (Reference 10). The latter models include: the Generalized Poisson Model (Reference 3), Goel's NHPP Model (Reference 10), Brooks and Motley's Model (Reference 11), and Norman Schneidewind's Model (Reference 12).

These models were chosen from among the many proposed for their performance in comparative studies and their adaptability to handle data collected from various testing environments.

In the next section the program's goals are described, along with how the program has been structured to accomplish these goals. Using a sample data set, the last section of the paper demonstrates some of the capabilities of the program by demonstrating how one would perform a reliability analysis.

SMERFS' GOALS AND DESIGN

During the development stage of the SMERFS program, certain goals were established to increase the benefit of this software reliability program. These goals touch on both the maintenance and the anticipated use of the program, and can be summarized as follows:

1. Maintainability,
2. Providing a complete reliability analysis environment,
3. Interactive in nature,
4. Error detection capabilities, and
5. Machine transportability.

MAINTAINABILITY

Software reliability is a relatively new field and therefore subject to change. Because the field is still growing, the SMERFS code was required to be in an easily maintained and fully documented state. To satisfy the established goal for ease in code understanding and alterations, all coding was performed in adherence to a Naval Surface Weapons Center (NSWC) publication on structured programming standards (Reference 13). This document directs code generation toward top-down design, indentations around loops and conditionals, and extensive in-line documentation. Additionally, the document requires that each routine of a program

contain prologue information of headings intended to provide routine understanding. These headings include: author, purpose, description, restrictions, communications (files, globals, and parameters), local glossary, errors, associated subprograms, references, language, declarations, and formats.

COMPLETE RELIABILITY ANALYSIS ENVIRONMENT

The second established goal addressed the completeness of the obtainable output from the SMERFS program. Besides the program including the eight models mentioned in the previous section, additional modules included: data input, data editing, transformations of the data, general summary statistics of the data, plots of the originally collected data, plots of the original and predicted values according to the fitted model, and a goodness-of-fit module to aid in determining the model adequacy (Figure 1). These various options are illustrated in the next section when a software reliability analysis is performed.

INTERACTIVE IN NATURE

The SMERFS program is designed to be flexible in execution. The program is made up of eight main modules (Figure 1). All but one of these modules have secondary modules or varying modes of execution. Because of the program's flexibility, the third established goal was that the program had to be utilized under an interactive mode. Under this method of execution, the program supplies the user with various menus and questions and the user inputs a response via the terminal keyboard. Free-format input of user responses was elected to reduce potential operational errors. This established goal generated other considerations in the program's design. The first was that the user should have complete control in the direction of the program. Reexecution of modules or omission of modules is directed solely by user responses. A second consideration was that the program had to load into a Control Data Corporation (CDC) 6700 computer in a reduced field length of 60K. This is a restriction imposed upon terminal executed programs at NSWC. This restriction was challenging to meet due to the massiveness of the error collection data base and the software package utilized for the graph generations. To satisfy this load length, the SMERFS program was written utilizing the CDC OVERLAY capability with one common data results vector and one temporary storage file.

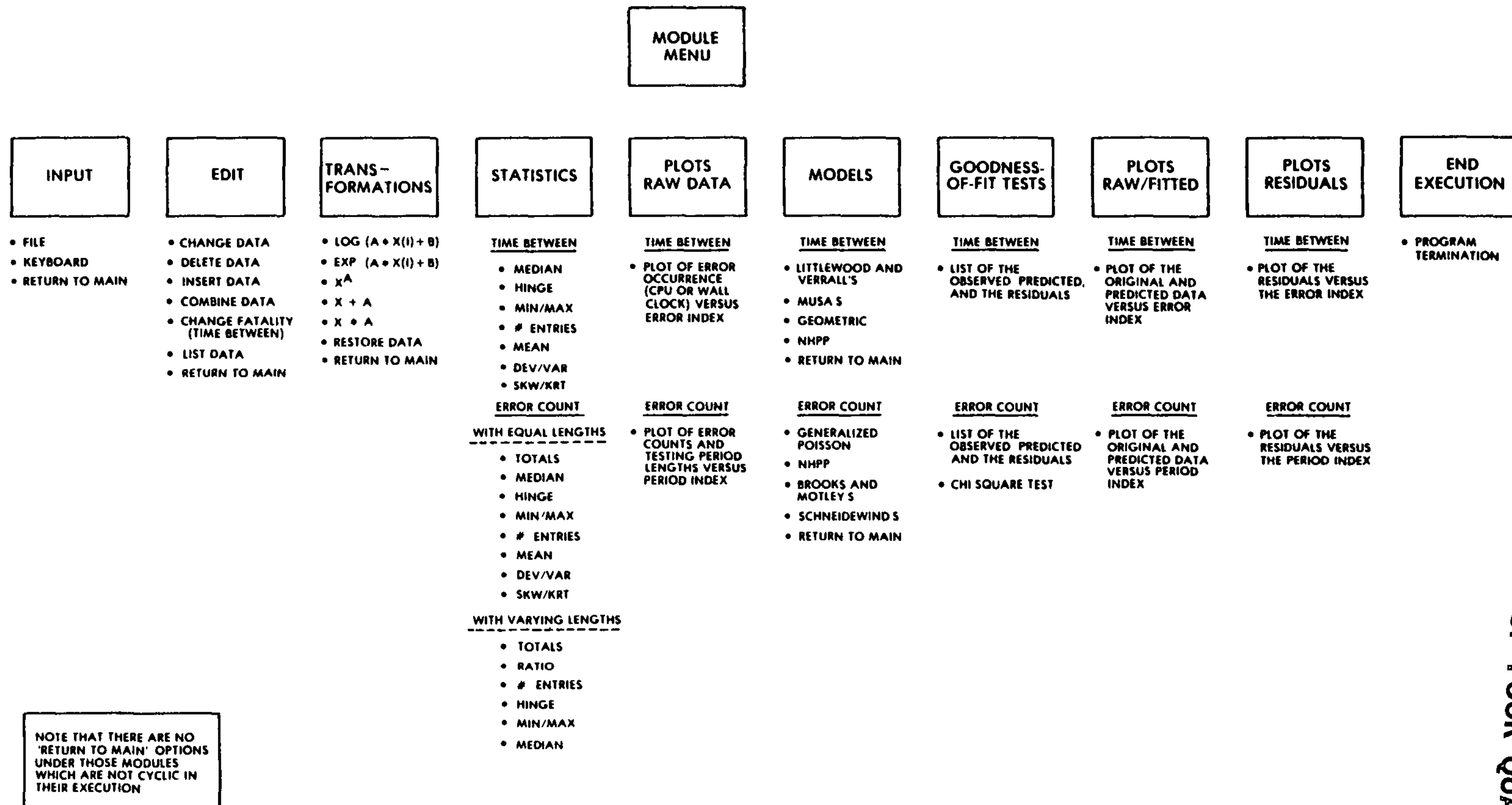


FIGURE 1. PROGRAM STRUCTURE

ORIGINAL PAGE IS
OF POOR QUALITY

ERROR DETECTION CAPABILITIES

The third established goal was that the program had to have complete error detection code in place. This meant it was designed with the capability to issue an informative error message and continue execution in a direction specified by the user, if either the user input an illegal response to a prompt or the numerical procedure to find the estimates of the model became unstable.

MACHINE TRANSPORTABILITY

The fifth and final goal addressed the potential for complete machine transportability of the code. The code of the software was developed in strict adherence to ANSI approved FORTRAN IV statements, with the exception of the following three areas: the CDC program card of file management, the use of free-format input, and the use of CDC OVERLAYS. The complete software operates on a CDC 6700 computer with a SCOPE 3.4 operating system. To allow for more machine transportability, the actual processing code of the software was removed and placed in a library. This created library is comprised completely of ANSI approved FORTRAN statements and therefore almost all facilities can utilize this library through simple CALL statements. The remaining portion of the program, known as the "DRIVER," consists of the input and output portions having the non-ANSI approved FORTRAN statements. Users with different computer systems, therefore, may only have to alter (or rewrite) this section of the program. However, full use of the software reliability library can be made.

SAMPLE DATA ANALYSIS

This section illustrates the use of the program in performing a reliability analysis on a set of data. The data were simulated on a computer and represent the number of errors detected per testing period. Each testing period was standardized to be one unit of length (1 day, 1 week, 1 month, etc.). The data were simulated to follow a non-homogenous Poisson process which satisfies the assumptions of Goel's NHPP Model. Since error count data are used in this example, none of the features of the program as applied to time between error detections are illustrated. Also, not all of the options provided by the program are illustrated, including aspects of data entry, data transformations, model fitting, and error detection within the program itself.

Figure 2 shows the menu that is provided to the user when the program is first executed. The various module options are listed in the order in which an analysis would be performed. The first chosen option would be DATA INPUT. The program then provides a menu showing the various input options (Figure 3). The program allows a preexisting data file to be entered (FILE INPUT), the data to be entered via a terminal keyboard (KEYBOARD INPUT), or a combination of both (FILE INPUT followed by KEYBOARD INPUT). If the KEYBOARD option is chosen, the program then asks for the type of data to be entered. The various options reflect the different data requirements of the various models (time between error occurrences as measured by elapsed wall clock and/or CPU time or error counts per testing interval). Since our example is error counts, the program prompts the user for the number of errors detected per period and the length of the period until the user is finished with data entry. This is indicated to the program with the entry of any negative numbers for the count and length. The user can then return to the main menu to pick the next module option.

If an error had been made in the data entry or a software error was subsequently analyzed not to be a programming error (e.g., an operator error), this necessitates a change to the error counts. The DATA EDIT option can be used to accomplish the required modifications. If the data need to be transformed in some manner, the DATA TRANSFORMATIONS option provides the user with a large selection of available transformations.

The user can next obtain various summary statistics pertaining to the entered data. These include: the median error count, the mean, the variance and standard deviation, the skewness and kurtosis measures for the data, and the number of errors discovered up to this point (Figure 4).

Module option 5 (PLOTS OF THE DATA) can be selected to provide either a plot of the raw data or a smoothed version of it. Figure 5 shows the plots provided for the sample data. The top plot is the raw error counts per testing period plotted against the testing period number. The smaller bottom plot represents testing period length versus period number. Notice the general downward trend exhibited by the data in the top plot. This indicates that fewer errors are being detected as testing progresses, thus indicating increasing reliability of the program.

```
SMERFS OUTPUT.    DATE: 10/04/84    TIME: 08.51.19.

PLEASE ENTER MODULE OPTION, ZERO FOR LIST=[0]
THE AVAILABLE MODULE OPTIONS ARE
 1 DATA INPUT
 2 DATA EDIT
 3 DATA TRANSFORMATIONS
 4 STATISTICS OF THE DATA
 5 PLOT(S) OF THE RAW DATA
 6 EXECUTION OF THE MODELS
 7 GOODNESS-OF-FIT TESTS
 8 PLOT OF ORIGINAL AND PREDICTED DATA
 9 PLOT OF RESIDUAL DATA
10 STOP EXECUTION OF SMERFS
PLEASE ENTER MODULE OPTION=[1]
```

NOTE: Blocked entries represent user input.

FIGURE 2. PROGRAM MENU

```
PLEASE ENTER INPUT OPTION, ZERO FOR LIST=[0]
THE AVAILABLE INPUT OPTIONS ARE
 1 FILE INPUT
 2 KEYBOARD INPUT
 3 RETURN TO THE MAIN PROGRAM
PLEASE ENTER INPUT OPTION=[2]
PLEASE ENTER KEYBOARD OPTION, ZERO FOR LIST=[0]
THE AVAILABLE KEYBOARD INPUT OPTIONS ARE
 1 WALL CLOCK TIME-BETWEEN-ERROR (WC TBE)
 2 CENTRAL PROCESSING UNITS (CPU) TBE
 3 WC TBE AND CPU TBE
 4 INTERVAL COUNTS AND LENGTHS
 5 RETURN TO THE INPUT ROUTINE
PLEASE ENTER KEYBOARD INPUT OPTION=[4]
 A RESPONSE OF NEGATIVE VALUES FOR THE PROMPT
 "PLEASE ENTER ERROR COUNT AND TEST LENGTH="
 WILL STOP PROCESSING

 PLEASE ENTER ERROR COUNT AND TEST LENGTH=[ 9][ 1]
 PLEASE ENTER ERROR COUNT AND TEST LENGTH=[15][ 1]
 PLEASE ENTER ERROR COUNT AND TEST LENGTH=[ 9][ 1]
 PLEASE ENTER ERROR COUNT AND TEST LENGTH=[13][ 1]
 PLEASE ENTER ERROR COUNT AND TEST LENGTH=[ 9][ 1]

                    •
                    •
                    •

 PLEASE ENTER ERROR COUNT AND TEST LENGTH=[ 3][ 1]
 PLEASE ENTER ERROR COUNT AND TEST LENGTH=[ 3][ 1]
 PLEASE ENTER ERROR COUNT AND TEST LENGTH=[ 3][ 1]
 PLEASE ENTER ERROR COUNT AND TEST LENGTH=[ 5][ 1]
 PLEASE ENTER ERROR COUNT AND TEST LENGTH=[-1][-1]

PLEASE ENTER INPUT OPTION, ZERO FOR LIST=[3]
```

NOTE: Blocked entries represent user input.

FIGURE 3. DATA INPUT

```
PLEASE ENTER MODULE OPTION, ZERO FOR LIST= [0]
THE AVAILABLE MODULE OPTIONS ARE
 1 DATA INPUT
 2 DATA EDIT
 3 DATA TRANSFORMATIONS
 4 STATISTICS OF THE DATA
 5 PLOT(S) OF THE RAW DATA
 6 EXECUTION OF THE MODELS
 7 GOODNESS-OF-FIT TESTS
 8 PLOT OF ORIGINAL AND PREDICTED DATA
 9 PLOT OF RESIDUAL DATA
10 STOP EXECUTION OF SMERFS
PLEASE ENTER MODULE OPTION= [4]

  INTERVAL DATA WITH EQUAL LENGTHS
   STATISTICS FOR ERROR COUNTS TOTALING TO    189
          **************************************
MEDIAN    *            .60000000E+01            *
HINGE     *    .40000000E+01     .90000000E+01  *
MIN/MAX   *    .20000000E+01     .15000000E+02  *
# ENTRIES *               28                    *
MEAN      *            .67500000E+01            *
DEV/VAR   *    .34278273E+01     .11750000E+02  *
SKW/KRT   *    .53692710E+00    -.45801780E+00  *
          **************************************

PLEASE ENTER MODULE OPTION, ZERO FOR LIST= [5]
```

NOTE: Blocked entries represent user input.

FIGURE 4. SUMMARY STATISTICS

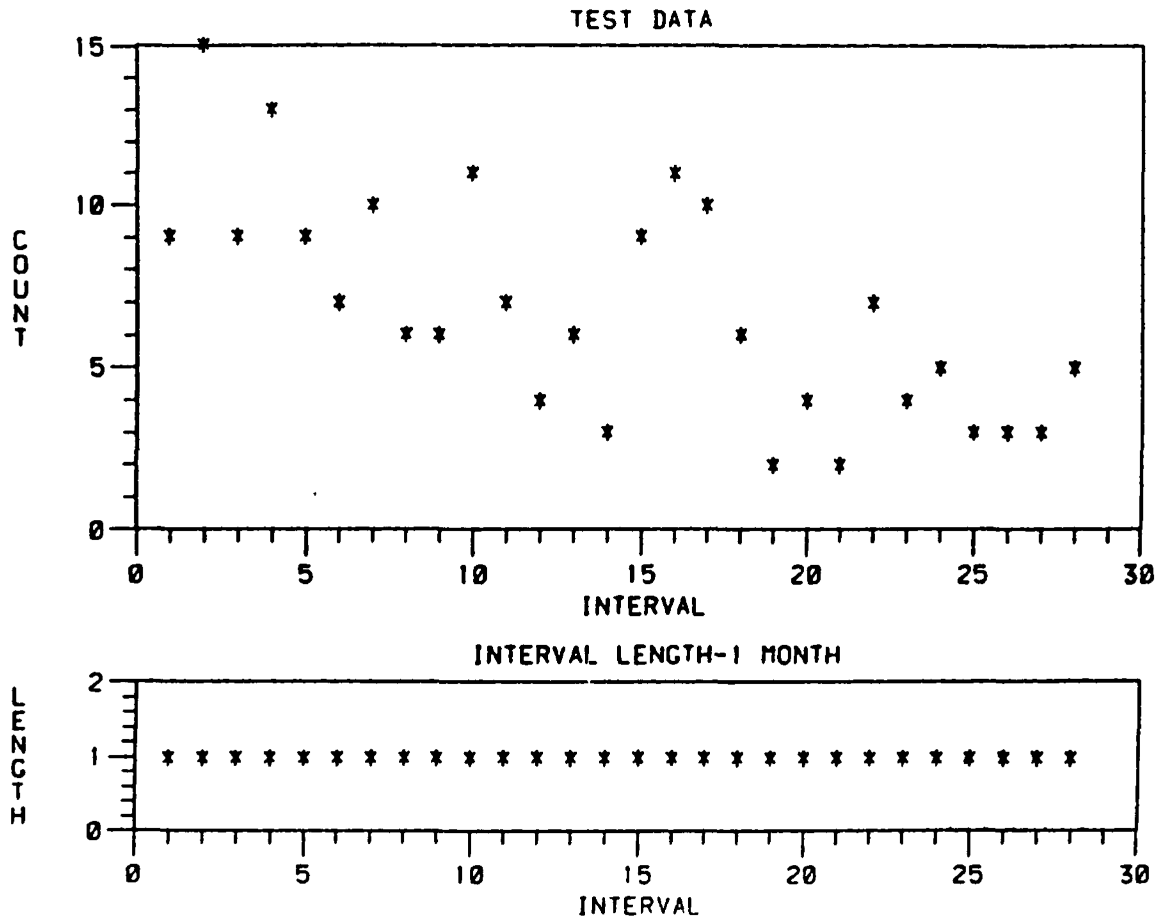


FIGURE 5. PLOTS OF RAW DATA

Module option 6 (EXECUTION OF THE MODELS) is next chosen for the actual model fitting. As Figure 6 indicates, a menu appropriate to the type of data entered is provided. The choice for this example execution is to fit the Non-Homogeneous Poisson Model. If the user desires, a list of the model assumptions and data requirements is provided to allow the user to make a judgment on the applicability of the model. If the user decides to continue with the candidate model, the program will request the number of iterations to be used in the numerical procedure and a starting value for that procedure. If the optimization procedure is successful, the various reliability estimates and corresponding precision of those estimates will be provided. In addition, the program will allow the user to iterate again to investigate the optimality of the derived estimates. In Figure 7, after 2 iterations, the maximum likelihood estimate of the proportionality constant in the NHPP Model was obtained as .043 with an associated 95% confidence interval of (.025, .061) and an estimate of the total number of errors residing in the code being 270 with a 95% confidence interval of (200, 340). The actual underlying parameters used to generate this data set were .05 and 250. The program for this particular model will allow the user, if desired, to estimate the number of errors expected in the next testing period. In this example, for an additional unit of testing, an additional four errors will be detected. Least squares estimates are also provided. These estimates (.043 and 269) are very close to the maximum likelihood ones.

After fitting a candidate model, the user can make a determination of the adequacy of the model by using options 7 (GOODNESS-OF-FIT TESTS), 8 (PLOT OF ORIGINAL AND PREDICTED DATA), and 9 (PLOT OF RESIDUAL DATA). Option 7 will perform a chi-square goodness-of-fit test as well as show a table of observed counts, predicted counts using the model, and the difference between the observed and the predicted (the residuals). For our example (Figure 8), the value of the chi-square statistic was 25.1 with an associated degrees-of-freedom of 25. If a test of hypothesis is made that the data set follows the candidate model, using an α-level of .05, the hypothesis would be accepted. Using option 8, the user can observe the raw and fitted model together (Figure 9). Option 9 (Figure 10) allows a plot of the residuals to aid in discovering any inadequacies in the model. Based upon the results of options 7 through 9, it appears that the NHPP Model can be used to estimate the reliability of the given program.

```
PLEASE ENTER MODULE OPTION, ZERO FOR LIST=[0]
THE AVAILABLE MODULE OPTIONS ARE
 1 DATA INPUT
 2 DATA EDIT
 3 DATA TRANSFORMATIONS
 4 STATISTICS OF THE DATA
 5 PLOT(S) OF THE RAW DATA
 6 EXECUTION OF THE MODELS
 7 GOODNESS-OF-FIT TESTS
 8 PLOT OF ORIGINAL AND PREDICTED DATA
 9 PLOT OF RESIDUAL DATA
10 STOP EXECUTION OF SMERFS
PLEASE ENTER MODULE OPTION=[6]

 PLEASE ENTER COUNT MODEL OPTION, ZERO FOR LIST=[0]
 THE AVAILABLE ERROR COUNT MODELS ARE
  1 GENERALIZED POISSON MODEL
  2 NON-HOMOGENEOUS POISSON MODEL
  3 BROOKS AND MOTLEY'S MODEL
  4 SCHNEIDEWIND'S MODEL
  5 RETURN TO THE MAIN PROGRAM
 PLEASE ENTER MODEL OPTION=[2]
```

NOTE: Blocked entries represent user input.

FIGURE 6. SELECTING A MODEL

```
PLEASE ENTER A 1 FOR MAXIMUM LIKELIHOOD, A 2 FOR LEAST
SQUARES, OR A 3 TO TERMINATE MODEL EXECUTION= [1]
PLEASE ENTER AN INITIAL ESTIMATE FOR THE PROPORTIONALITY CONSTANT
 (A NUMBER BETWEEN ZERO AND ONE)= [0.04]
PLEASE ENTER THE MAXIMUM NUMBER OF ITERATIONS= [100]

ML MODEL ESTIMATES AFTER  2 ITERATIONS ARE:
 PROPORTIONALITY CONSTANT OF THE MODEL IS  .43140563E-01
  WITH APP. 95% C.I. OF (  .24941691E-01,  .61339435E-01)
 THE TOTAL NUMBER OF ERRORS IS             .26954311E+03
  WITH APP. 95% C.I. OF (  .19963048E+03,  .33945575E+03)

PLEASE ENTER 1 FOR AN ESTIMATE OF THE NUMBER OF ERRORS
EXPECTED IN THE NEXT TESTING PERIOD; ELSE ZERO= [1]
 PLEASE ENTER THE PROJECTED LENGTH OF THE TESTING PERIOD= [1]
 THE EXPECTED NUMBER OF ERRORS IS  .34007917E+01

PLEASE ENTER A 1 FOR MAXIMUM LIKELIHOOD, A 2 FOR LEAST
SQUARES, OR A 3 TO TERMINATE MODEL EXECUTION= [2]
PLEASE ENTER AN INITIAL ESTIMATE FOR THE PROPORTIONALITY CONSTANT
 (A NUMBER BETWEEN ZERO AND ONE)= [0.043]
PLEASE ENTER THE MAXIMUM NUMBER OF ITERATIONS= [100]

LS MODEL ESTIMATES AFTER  2 ITERATIONS ARE:
 PROPORTIONALITY CONSTANT OF THE MODEL IS  .43315840E-01
 THE TOTAL NUMBER OF ERRORS IS             .26890859E+03

PLEASE ENTER 1 FOR AN ESTIMATE OF THE NUMBER OF ERRORS
EXPECTED IN THE NEXT TESTING PERIOD, ELSE ZERO= [1]
 PLEASE ENTER THE PROJECTED LENGTH OF THE TESTING PERIOD= [1]
 THE EXPECTED NUMBER OF ERRORS IS  .33895981E+01

PLEASE ENTER A 1 FOR MAXIMUM LIKELIHOOD, A 2 FOR LEAST
SQUARES, OR A 3 TO TERMINATE MODEL EXECUTION= [3]
```

NOTE: Blocked entries represent user input.

FIGURE 7. MODEL ESTIMATION PROCEDURES

```
PLEASE ENTER MODULE OPTION, ZERO FOR LIST=[7]

  PLEASE ENTER THE CELL COMBINATION FREQUENCY (THE STANDARD
  IS A FIVE); OR A MINUS 1 TO INDICATE NO CELL COMBINATIONS= [-1]
   THE CHI-SQUARE STATISTIC IS  .25055379E+02
   WITH   25 DEGREES-OF-FREEDOM
   PLEASE ENTER 1 TO TRY ANOTHER COMBINATION FREQUENCY; ELSE ZERO=[0]

  PLEASE ENTER 1 FOR THE DATA LISTING; ELSE ZERO=[1]


  NUMBER  ORIGINAL  DATA    PREDICTED DATA    RESIDUAL  DATA
  ======  ==============    ==============    ==============
       1   .90000000E+01     .11380985E+02    -.23809854E+01
       2   .15000000E+02     .10900443E+02     .40995567E+01
       3   .90000000E+01     .10440191E+02    -.14401912E+01
       4   .13000000E+02     .99993724E+01     .30006276E+01
       5   .90000000E+01     .95771664E+01    -.57716642E+00
       6   .70000000E+01     .91727874E+01    -.21727874E+01
       7   .10000000E+02     .87854825E+01     .12145175E+01
       8   .60000000E+01     .84145309E+01    -.24145309E+01
       9   .60000000E+01     .80592421E+01    -.20592421E+01
      10   .11000000E+02     .77189547E+01     .32810453E+01
      11   .70000000E+01     .73930354E+01    -.39303536E+00
      12   .40000000E+01     .70808774E+01    -.30808774E+01
      13   .60000000E+01      67818998E+01    -.78189976E+00
      14   .30000000E+01     .64955459E+01    -.34955459E+01
      15   .90000000E+01     .62212829E+01     .27787171E+01
      16   .11000000E+02     .59586001E+01     .50413999E+01
      17   .10000000E+02     .57070087E+01     .42929913E+01
      18   .60000000E+01     .54660402E+01     .53395976E+00
      19   .20000000E+01     .52352463E+01    -.32352463E+01
      20   .40000000E+01     .50141972E+01    -.10141972E+01
      21   .20000000E+01     .48024815E+01    -.28024815E+01
      22   .70000000E+01     .45997051E+01     .24002949E+01
      23   .40000000E+01     .44054906E+01    -.40549063E+00
      24   .50000000E+01     .42194765E+01     .78052349E+00
      25   .30000000E+01     .40413165E+01    -.10413165E+01
      26   .30000000E+01     .38706790E+01    -.87067900E+00
      27   .30000000E+01     .37072464E+01    -.70724637E+00
      28   .50000000E+01     .35507144E+01     .14492856E+01

PLEASE ENTER MODULE OPTION, ZERO FOR LIST= [8]
```

NOTE: Blocked entries represent user input.

FIGURE 8. GOODNESS-OF-FIT

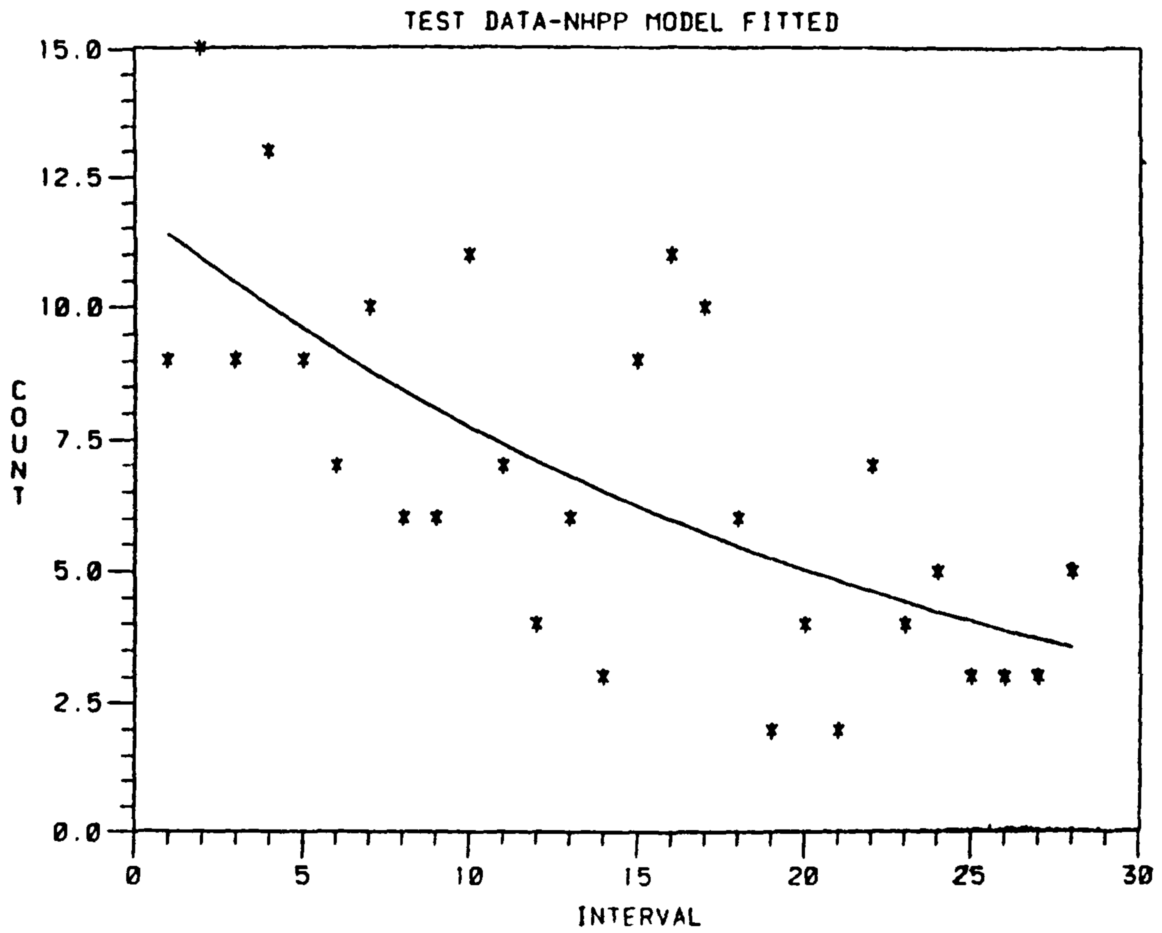


FIGURE 9. MODEL FIT OF DATA

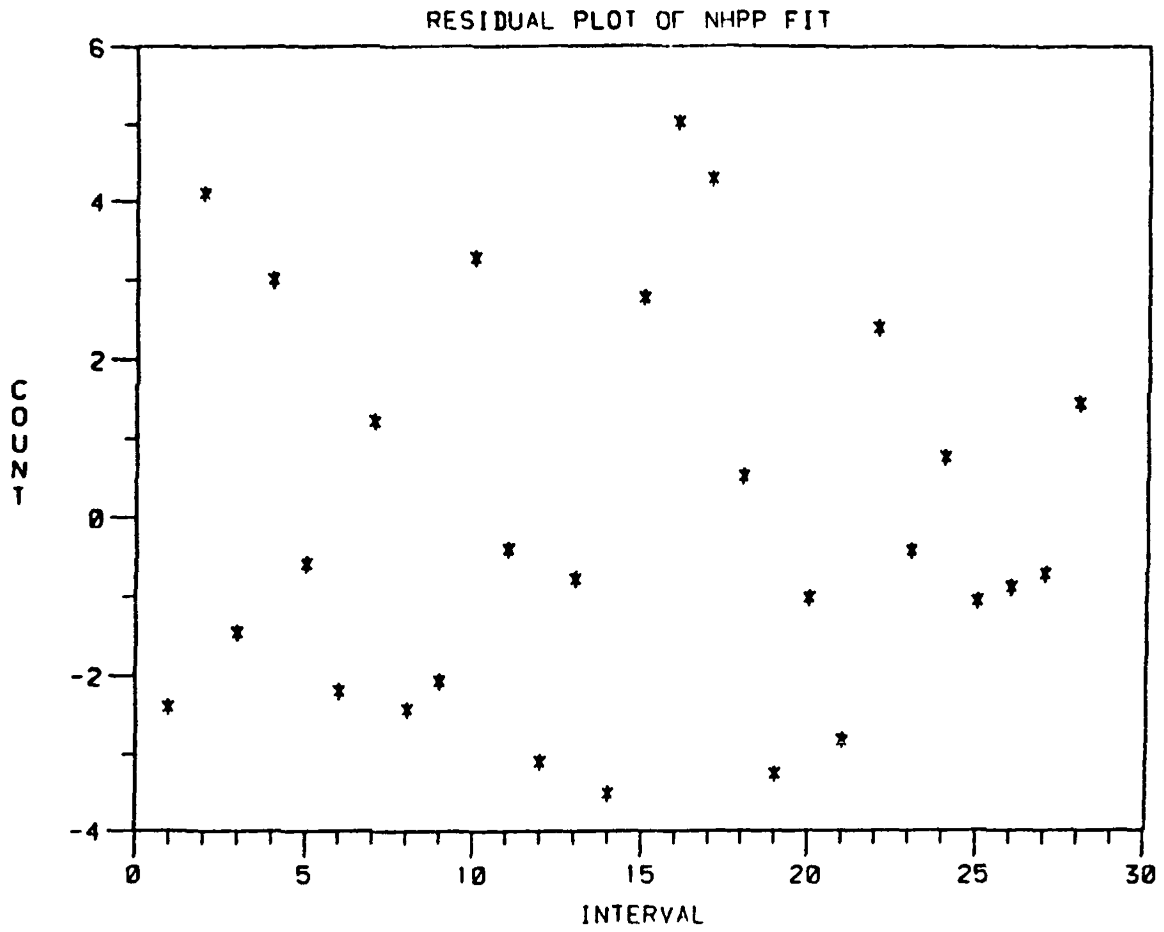


FIGURE 10. PLOT OF RESIDUALS

If the model was inadequate the user could have tried an alternative model, and/or transformed the data before fitting the model. The interactive capability of the program allows the user to dynamically create the best model for the given set of data.

SUMMARY

With the rapid growth of computer software, researchers have been developing tools and techniques which will aid in developing reliable software. One such area has been the estimation of a program's reliability using past error discovery data. Many different models have been proposed using these data to estimate various measures of reliability (total number of errors, expected time until the next error, etc.). These models, however, require sophisticated numerical procedures to obtain the estimates, necessitating the use of the computer. An interactive computer program, SMERFS, has been developed which allows the user to enter a set of data, modify it if necessary, fit an appropriate model, and determine the adequacy of the fitted model. This tool allows rapid assessment of a program's reliability during the testing phase. This, in turn, helps in addressing the age old question, "How do I know when the software should be released?".

REFERENCES

1. M. Shooman, "Software Reliability: Analysis and Prediction," Integrity in Electronic Flight Control Systems, AGARDograph No. 224, Advisory Group for Aerospace Research and Development, Part II, p. 7, 1977.

2. L. S. Gephart, C. M. Greenwald, M. M. Hoffman, and D. H. Osterfeld, Software Reliability: Determination and Prediction, Air Force Flight Dynamics Laboratory Technical Report, AFFDL-TR-78-UU, June 1978.

3. R. E. Schafer, J. F. Alten, J. E. Angus, and S. E. Emota, Validation of Software Reliability Models, Rome Air Development Center Technical Report, RADC-TR-79-147, 1979.

4. George J. Shick and Ray W. Wolverton, "An Analysis of Competing Software Reliability Models," IEEE Transactions on Software Engineering, Vol. SE-4, No. 2, March 1978, pp. 104-120.

5. C. V. Ramamoorthy and F. B. Bastani, "Software Reliability - Status and Perspectives," IEEE Transactions on Software Engineering, Vol. SE-8, No. 4, July 1982, pp. 354-371.

6. W. H. Farr, A Survey of Software Reliability Modeling and Estimation, Naval Surface Weapons Center Technical Report, NSWC TR 82-171, June 1983.

7. B. Littlewood and J. Verrall, "A Bayesian Reliability Growth Model for Computer Software," The Journal of the Royal Statistical Society, Series C, Vol. 22, No. 3, 1973, pp. 332-346.

8. P. Moranda, "Predictions of Software Reliability During Debugging," 1975 Proceedings of the Annual Reliability and Maintainability Symposium, Washington, DC, 1975.

9. J. Musa, "A Theory of Software Reliability and Its Applications," IEEE Transactions on Software Engineering, Vol. SE-1, No. 3, September 1975, pp. 312-327.

REFERENCES (Cont.)

10. A. Goel and K. Okumoto, "Time-Dependent Error-Detection Rate Model for Software Reliability and Other Performance Measures," IEEE Transactions on Reliability, Vol. R-28, No. 3, August 1979, pp. 206-211.

11. W. D. Brooks and R. W. Motley, Analysis of Discrete Software Reliability Models, Rome Air Development Center Technical Report, RADC-TR-80-84, April 1980.

12. N. F. Schneidewind, "Analysis of Error Processes in Computer Software," Sigslan Not., Vol. 10, No. 6, 1975, pp. 337-346.

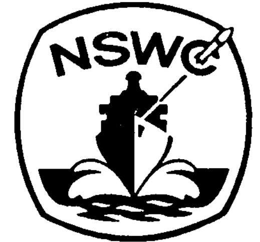
13. R. T. Bevan and J. H. Reynolds, Computer Programming and Coding Standards for the FORTRAN and SIMSCRIPT II.5 Programming Languages, Naval Surface Weapons Center Technical Report, NSWC TR-3878, December 1981.

THE VIEWGRAPH MATERIALS

for the

W. FARR PRESENTATION FOLLOW

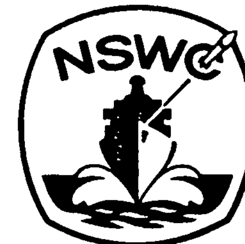
AN INTERACTIVE PROGRAM FOR SOFTWARE RELIABILITY MODELING



WILLIAM FARR – NSWC

OLIVER SMITH – EG&G

OUTLINE OF PRESENTATION

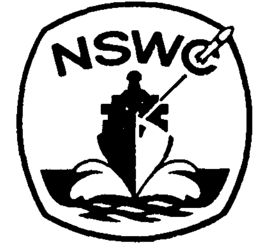


PROGRAM GOALS AND DESIGN

SAMPLE DATA ANALYSIS

CONCLUSIONS

PROGRAM GOALS AND DESIGN



MAINTAINABILITY

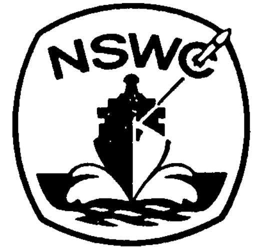
- STATE-OF-THE-ART PROGRAMMING CONVENTIONS AND STRUCTURES

COMPLETE RELIABILITY ANALYSIS

- INTERACTIVE IN EXECUTION WITH ERROR DETECTION AND CORRECTION

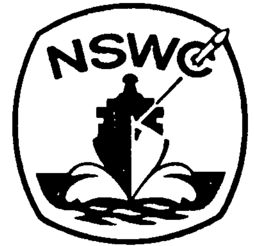
MACHINE TRANSPORTABILITY

- DRIVER – CONTAINING I/O
- LIBRARY – CONTAINING COMPUTATIONS



SAMPLE DATA ANALYSIS

MODULE MENU



DATA INPUT

DATA EDIT

DATA TRANSFORMATIONS

STATISTICS OF THE DATA

PLOT(S) OF THE RAW DATA

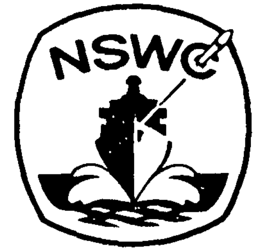
EXECUTION OF THE MODELS

GOODNESS-OF-FIT TESTS

PLOT OF ORIGINAL AND PREDICTED DATA

PLOT OF RESIDUAL DATA

INPUT MODULE



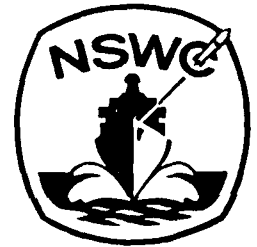
TYPES OF INPUT

- FILE
- KEYBOARD

TYPES OF DATA

- TIME BETWEEN ERROR
  - WALL CLOCK UNITS (24 HR.)
  - CPU
- ERROR COUNTS
  - ERROR COUNTS PER TESTING PERIOD
  - TESTING PERIOD LENGTHS

MODELS MODULE



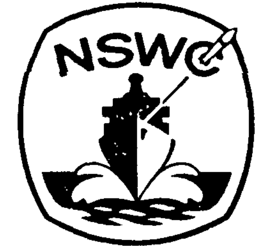
TIME BETWEEN ERROR MODELS

- LITTLEWOOD AND VERRALL'S BAYESIAN MODEL
- MUSA'S EXECUTION TIME MODEL
- GEOMETRIC MODEL
- NON-HOMOGENEOUS POISSON EXECUTION TIME MODEL

ERROR COUNT MODELS

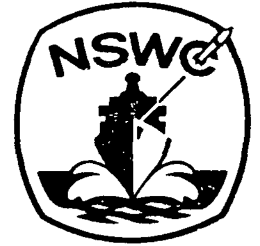
- GENERALIZED POISSON MODEL
- NON-HOMOGENEOUS POISSON INTERVAL DATA MODEL
- BROOKS AND MOTLEY'S DISCRETE MODEL
- SCHNEIDEWIND'S MAXIMUM LIKELIHOOD MODEL

RESULTS OF THE NHPP MODEL FIT



	MAXIMUM LIKELIHOOD	LEAST SQUARES
TOTAL NUMBER OF ERRORS	269.5 (199.6,339.5)	268.9
PROPORTIONALITY CONSTANT	0.043 (0.025, 0.061)	0.043
PREDICTED NUMBER OF ERRORS IN THE NEXT PERIOD	3.4	3.4

PLOT OF RAW DATA AND FITTED MODEL



TEST DATA

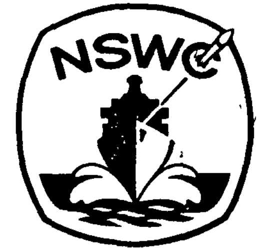
COUNT

15.0
12.5
10.0
7.5
5.0
2.5
0.0

0 5 10 15 20 25 30

INTERVAL

CONCLUSIONS



COMPLETE RELIABILITY ANALYSIS

MODULARITY IN DESIGN

FULLY DOCUMENTED

N86-19979

ASSESSING THE PROFICIENCY OF SOFTWARE DEVELOPERS ©

by

Lawrence H. Putnam
Douglas T. Putnam
Lauren P. Thayer

In the mid 1970's Lawrence Putnam developed a equation that explained the behavior of software systems. He called it the software equation. It is written in the form:

$$Ss = Ck \times K^{1/3} \times Td^{4/3}, \text{ Subject to } K/Td^3 \leq G$$

Notice that there are only four terms in the basic equation. The components are defined as:

Ss - The total number of DDESLOC **
Ck - An overall efficiency-complexity measure
K - Total Life-cycle effort
Td - The development time
G - maximum manpower acceleration possible
  for a class of system

Thus, a given product can be developed in Td amount of time, for K amount of effort, at C_k efficiency level.

The software equation can be thought of as a powerful trade-off law. A given product developed in a fixed environment, could be developed with many different time-effort combinations, all of which would satisfy the equation. However, because of the time and effort exponents, the equation gives dramatic results. With these exponents small changes in time produce substantial changes in effort. In practical use the software equation has demostrated it can be a high leverage software management function.

Over the past 5 years we have analysed data from over 2000 software projects. Our intention was to independently validate the software equation. Of those 2000 projects some 803 had complete data and have been entered into our database. With this data we have been able to prove that the exponents are very close to the true behavior.

** DDESLOC is the notation use for Delivered, Developed, Executable, Source Lines of Code.

(C) Copyright by Quantitative Software Management, Inc. April 1984

In late 1982 we established regression trend lines for our database. Regression lines were developed for the measures listed below.

Productivity (Ss/MM) vs DDESLOC
Schedule vs DDESLOC
Effort vs DDESLOC
Average Manpower (MM/MOS) vs DDESLOC

In the initial analysis we observed clusterings in the values of Ck. The cluster patterns were related to application type. It was thought that the trend lines might be correlated to Ck. Each application type should have it's own family of trend lines that would shift up or down according to the range of Ck values present.

By mid 1983 the database was large enough to stratify according to application type. The major categories identifed were:

Real time Embedded systems
Avionics systems
Management Information systems
Scientific systems
Command and Control systems
Systems software
Microcode and Firmware systems

The curve fitting exercise confirmed our thoughts. The trend lines did shift. Micro-code and firmware were located at the low end of the spectrum. This software had low values for Ck, low productivity, took a long time, was quite expensive and demanded more people relative to similar sized projects. The MIS application were at the high end of the spectrum. These systems had high values for Ck, high productivity, shorter schedules, were less expensive and used fewer people relative to their size.

Variability around the average trend lines was still a concern. Could the software equation explain that variability? There is a ratio that effectively measures the application of effort over time. This measure is called the Manpower Buildup Gradient. It is defined as K/Td^3. It discloses the style of the software development organization. High values (generally larger than 20) are present when parallel effort is possible and management is willing to commit whatever resources are necessary to get a system built fast. Low values are more typical of sequential efforts (design intensive processes) or a management constrainted situation (limited available manpower).

New data was analyzed using the new trend lines as a basis for comparison. We found that it told a consistent and unambigious story. The typical behavior pattern for systems with a steep manpower buildup rate is: modest schedule compression, lower productivity (Ss/MM), higher average code production (Ss/Mos), requiring more effort and more people. Conversely, systems with a gradual manpower buildup rate had slightly longer schedules, much higher productivity, lower average code

production rates, requiring less effort and fewer people.

CASE STUDY (A Major Computer Vendor)

An independent data set from a major U.S. computer manufacturer illustrates these points. The systems used in this analysis come from a manufacturing facility dedicated to building smart IBM compatible mainframe terminals. The software that drives the most recent family of terminals is written primarily in C language with a small portion of assembly code. The primary system functions are diagnostics, memory management, and communication. This family of products has a limited market share. The costs associated with product development are high but can be recovered along with a profit if the manufacturer can deliver the product within a narrow market window. The software is the guts of the product and therefore critically important. Company management is willing to dedicate large software development staffs to get whatever schedule compression is needed to meet the market demands (regardless of whether the schedule is realistic or not).

The data from three systems developed recently at this plant is summarized in the top portion of Table 1. Notice that two systems are RAM based. Due to a hardware constraint the third system had to be written so that it could reside in ROM. The unique problems present on the ROM development include severely limited memory and very high performance specifications. High quality was essential on the ROM system because it involved a manufacturing process and would be costly to replace once it was in the field.

The bottom portion of Table 1 summarizes important calculations made on the input data. The column titled Productivity Index uses a linear sequence of numbers that relate to the actual Ck values in parenthesis. Likewise the Manpower Buildup Index corresponds to the Manpower Buildup Gradient values in parenthesis. Notice that there is a big difference between the Ck values of the RAM and ROM based systems.

The Manpower Buildup Gradient for all three systems are high. The value calculated from RAM #2 is more than double that of RAM #1. According to the software trade-off law there should be a noticable difference between the two systems for the time and effort required to complete these projects. The other measures summarized in Table 1 are dependent on system size. Taken out of this context they are not meaningful. However, if we compare them against a baseline for their own size and application then they will be meaningful.

GRAPHICAL ANALYSIS (QSM System Software Database)

Figure 1 is a frequency graph of the Manpower Buildup Index for the three systems. Two observations can be made from this chart. The development style of this company is to staff up quickly and use alot

of people. This management practice can be explained by the market environment. A second observation is worthy of notice. The RAM based systems have different manpower buildup index measures. The data from RAM #1 calculates a 3. RAM #2 calculates a 4. The management trade-off decisions that produced these systems should show exchanges of time and effort according to the software equation.

Figure 2 show the distribution of Ck. The graph utilizes an index which the Ck values fall within. It is immediately obvious that there is a big difference between the two types of software implementations. The difference can not be attributed to the function that the software performs. They are quite similar. Rather, it is in the way the code has to be designed and written for the particular implementation which is quite different. The ROM software is more difficult because it requires designing tricky code overlays to meet memory restriction and needs constant performance tuning. It must be bug free before it is burned into ROM. RAM #2 has a higher Ck than RAM #1. With a higher Ck RAM #2 utilized less overall effort. The higher efficency of RAM #2 will counteract the nonlinear effort increase attributed to the steeper Manpower Buildup Rate that RAM #2 had.

In Figure 3, the data is superimposed on the average manpower trend lines for the System software database. Notice that the scales are logrithmic. The log scales turn the non-linear trends into straight lines. The absissa (X axis) represents the total number of Delivered Developed Executable Source Lines of Code. The ordinate (Y axis) is the average number of people (MM/Td). There are three trend lines drawn on the graph. The middle line is the best regression fit for all the data contained in the Systems software database. The high and low lines are the plus and minus one standard deviation bounds. Each cross represents the calculated average manpower plotted at the reported size.

The ROM system required significantly more people than other comparably sized systems software projects. On the other hand, the RAM based systems are very economic in their use of manpower compared to the industry average for their size. In a relative sense RAM #1 has a lower manpower utilization compared to RAM #2. This can be attributed to the more gradual manpower buildup rate.

Figure 4 is a similar portrayal of the database. In this case we will compare Total Manmonths against the system size. Since manmonths are proportional to cost, this graph compares these systems for cost effectiveness. The ROM system is significantly more expensive. The RAM systems are well below the industry average. RAM #1 appears to be a little less expensive compared to RAM #2.

Figure 5 compares the data against the productivity trend lines. The ROM system is close to two standard deviations lower than the average for that sized project. The RAM based systems again are better than the industry average. Notice that RAM #1 has a better relative postion compared to it's size.

Figure 6 starts to disclose the trade-off situation. This figure compares the data against the trend lines for average project

duration. It is no surprise that the ROM system took significantly longer than the average. The RAM systems are interesting. RAM #2 is somewhat shorter than the average. In constrast RAM #1 is a little longer than the average. The pattern seems to coming together. RAM #2 as you recall had the steeper manpower buildup rate. The objective must have been to get the system built fast. The system was built in a shorter time but in a relative sense it required a lot more effort and people. The difference would be more pronounced if the values of Ck were the same. The non-linearities present in software equation are still powerful enough counteract the lower efficency of RAM #1.

The pattern continues in Figure 7. The ROM system is again below the industry average. The Ck for the ROM system was well below the Systems software average and explains why it compares in such a unfavorable way. RAM #2 experienced a rapid manpower buildup and therefore had a higher code production rate. RAM #1 had a more gradual manpower buildup and a lower relative code production rate.

CONCLUSION

The trend lines presented in this paper can be useful in a number of ways. They provide a baseline of comparision from which software developers can compare their performance against a large database of similar projects. This will often identify a organizational style. In this case study it was possible to quantify the organizational style using the Manpower Buildup Gradient. Additionally we were able to show that the developer was a better than average producer on RAM based systems. The Ck associated with the ROM system suggests that it is a different class of work. When this system is compared against the Firmware database it is very creditable.

It is important to recognize that there are non-linearities present in the software process. The non-linearities are tied to system size. For comparative purposes we must always make judgements based on similar sizes. In the past the tendency has been to calculate a few ratios on several projects and then compare them without any regard to amount of functonality that was created. This practice can be very misleading and dangerous. The method describe in this paper used in a thoughtful analytic manner can be very helpful.

There are some problems associated with curve fitting that should be pointed out. With the non-linearities present in software, small data sets will often produce wide variations in slope. Any effort (MM) dependent ratios are particularly troublesome. Productivity has consistently proven to be the most sensitive. Of all fits on productivity (Ss/MM) that we have made we have never been able to get a r squared value better than (.02). The nonlinearities in the terms productivity is composed of are responsible for this. To work around this situation we have chosen to combine a theoretical slope tuned by the actual data.

It is possible to extend this approach. The present plans include providing for a reliability comparision. Right now the error database is not large enough to get totally reliable statistics but before too long we hope to establish those trend lines as well. The database will be analyzed to determine the improvement that is being made in each of the application areas over time. Some preliminary work in this area has been done and it looks very promising.

CALIBRATE INPUT SUMMARY

SYSTEM NAME	SIZE (SS)	TIME (MOS)	EFFORT (MM)	APPLICATION TYPE	OPERATIONAL DATE
RAM SOFTWARE #1	229000	24.0	869	SYSTEM SOFTWARE	
RAM SOFTWARE #2	45800	10.0	133	SYSTEM SOFTWARE	
ROM SOFTWARE #1	14000	16.0	178	SYSTEM SOFTWARE	

MANAGEMENT METRICS

SYSTEM NAME	PRODUCTIVITY INDEX C_k	MANPOWER BUILDUP INDEX G	PRODUCTIVITY (SS/MM)	PEAK MANPOWER (PEOPLE)	AVG CODE PRODUCTION RATE (SS/MO)
RAM SOFTWARE #1	12 (15,976)	3 (23)	264	56	9541
RAM SOFTWARE #2	14 (20,348)	4 (53)	374	19	4980
ROM SOFTWARE #1	5 (2,100)	3 (39)	79	17	875

Table 1

ORIGINAL PAGE IS OF POOR QUALITY

COMPUTER VENDOR
STAFFING BUILDUP DEV DATABASE

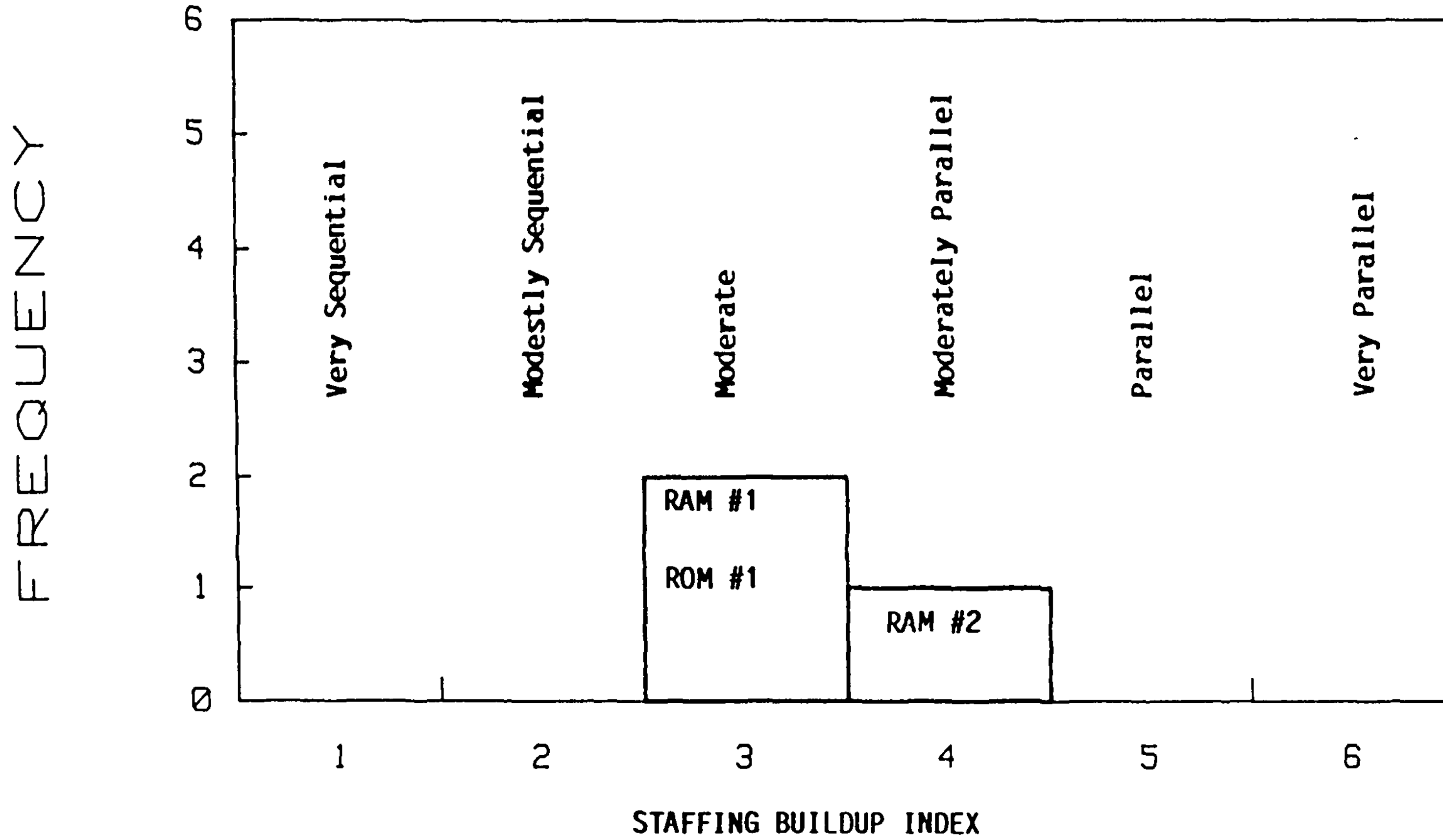


FIGURE 1

COMPUTER VENDOR TECHNOLOGY FACTOR DEV DATABASE

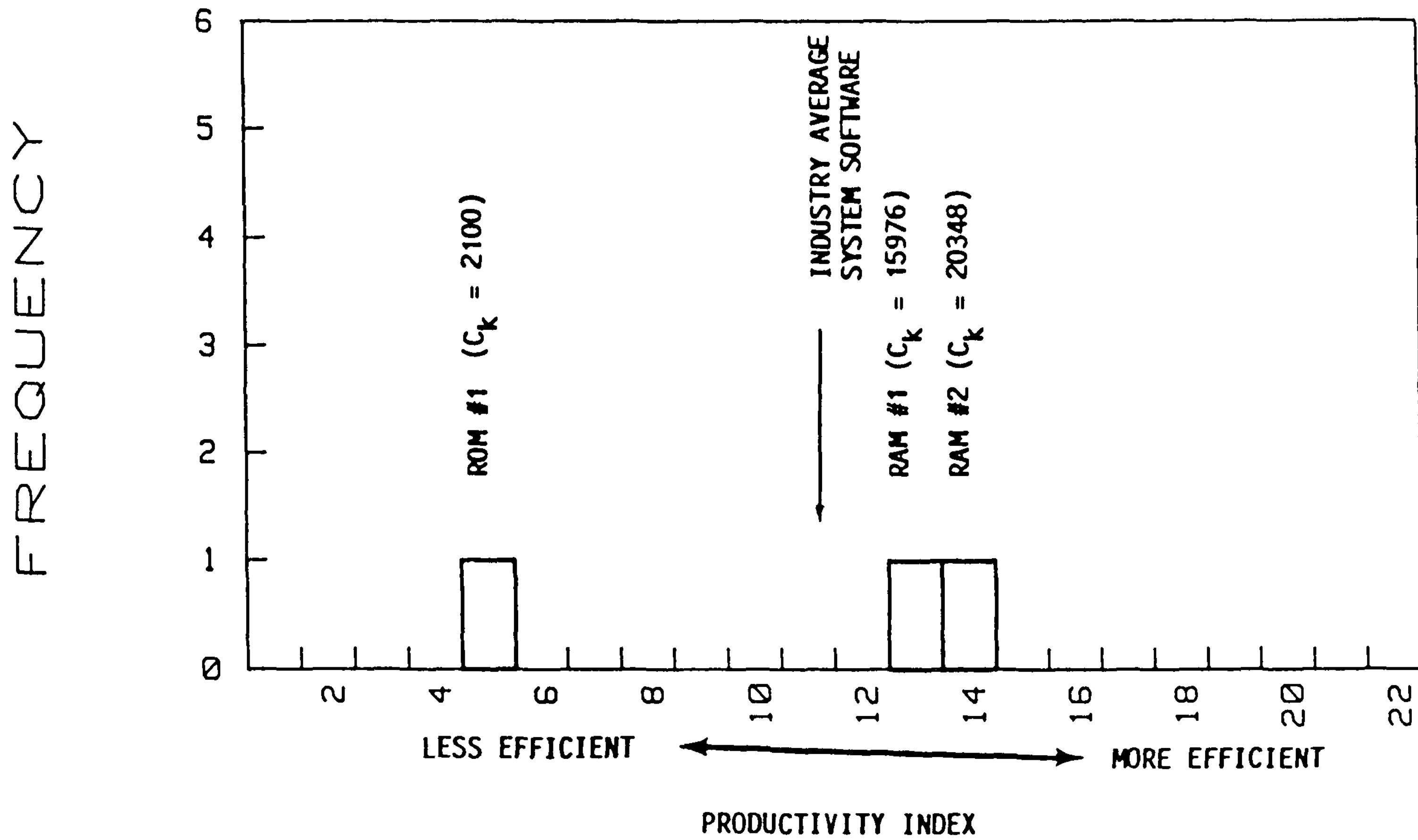


FIGURE 2

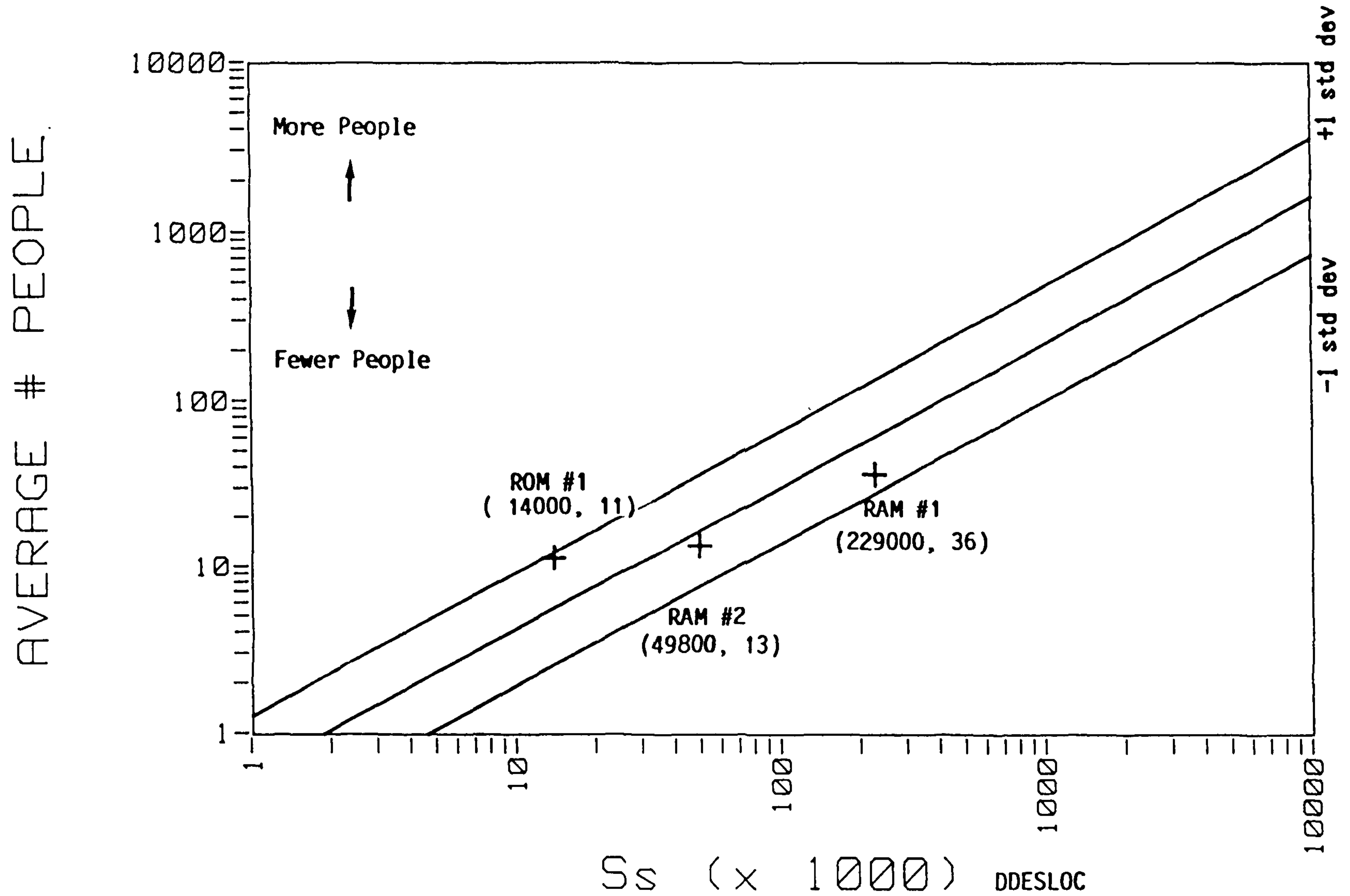


FIGURE 3

SYSTEM SFTW - TOTAL MANMONTHS
VS. Ss

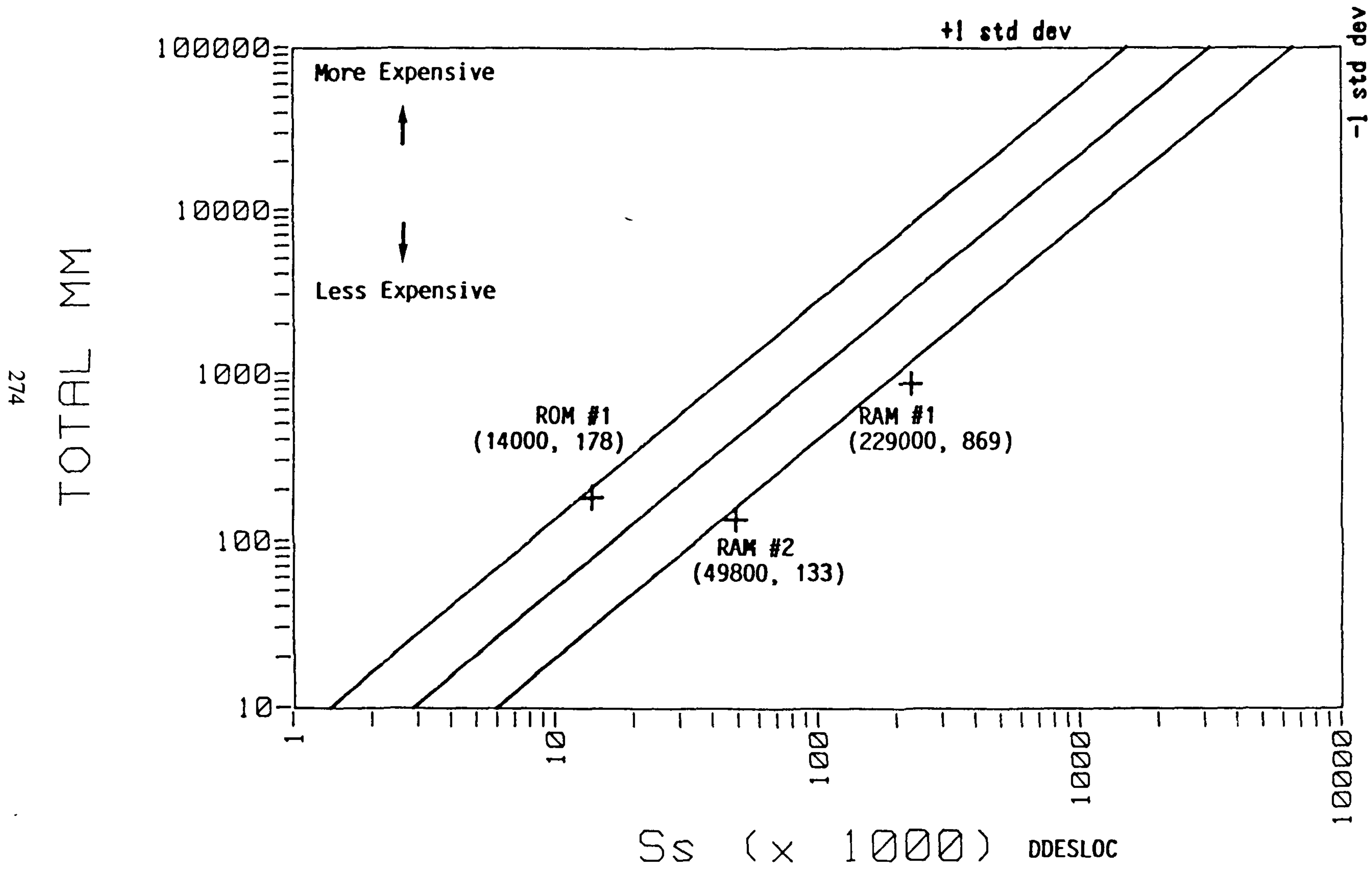


FIGURE 4

SYSTEM SFTW - PRODUCTIVITY
VS. Ss

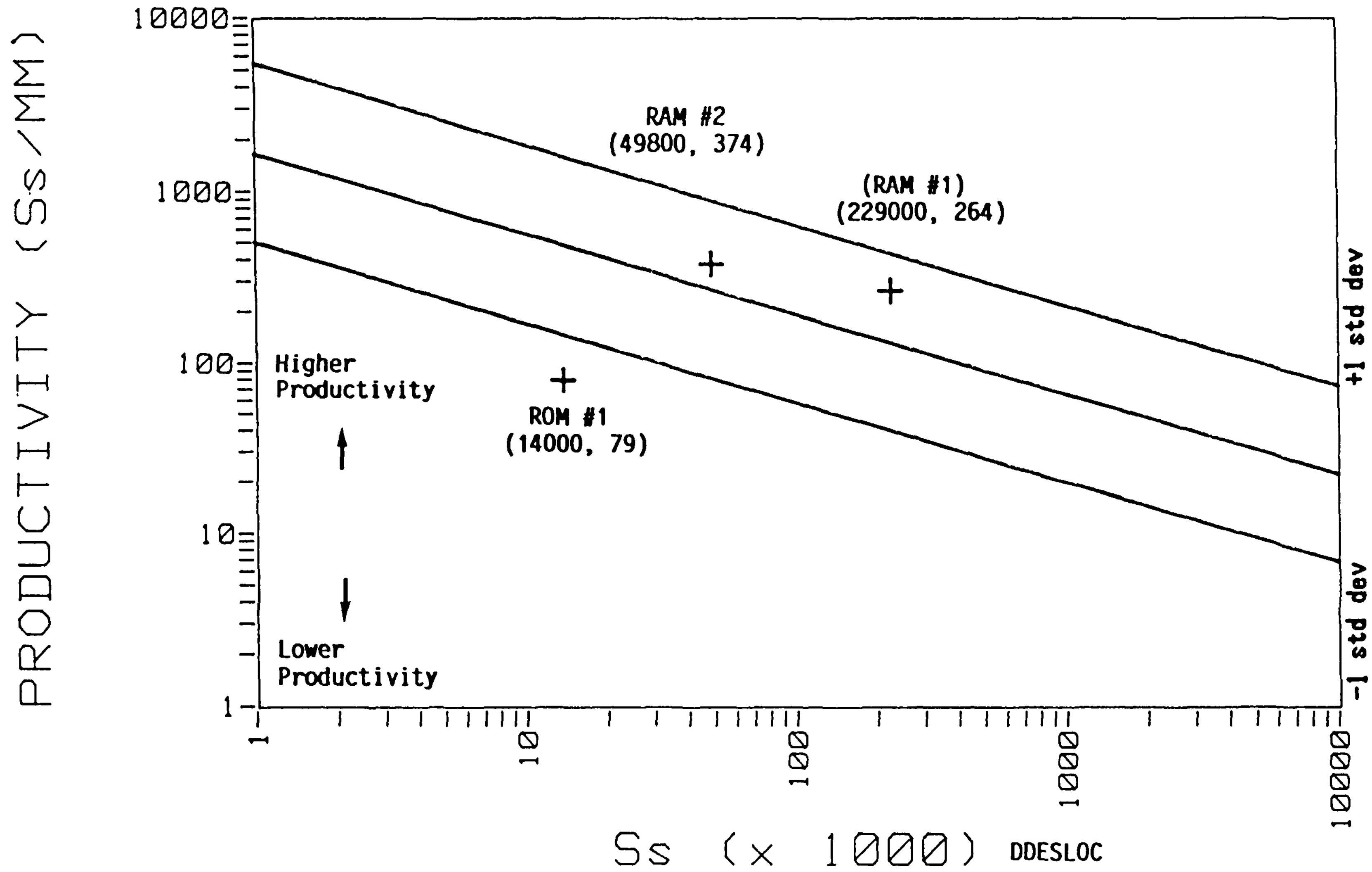


FIGURE 5

SYSTEM SFTW - PROJECT DURATION (Mos)
VS. Ss

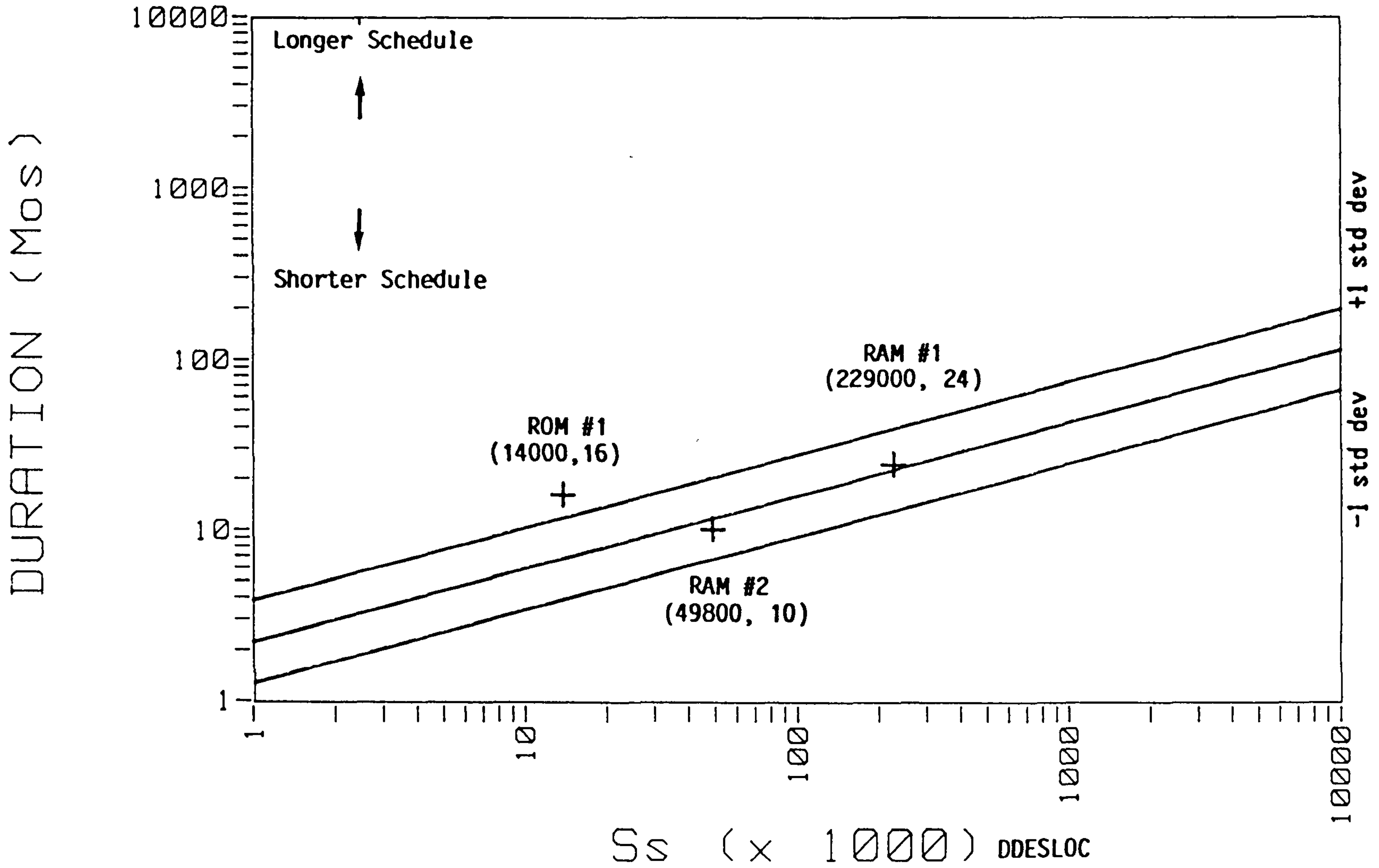


FIGURE 6

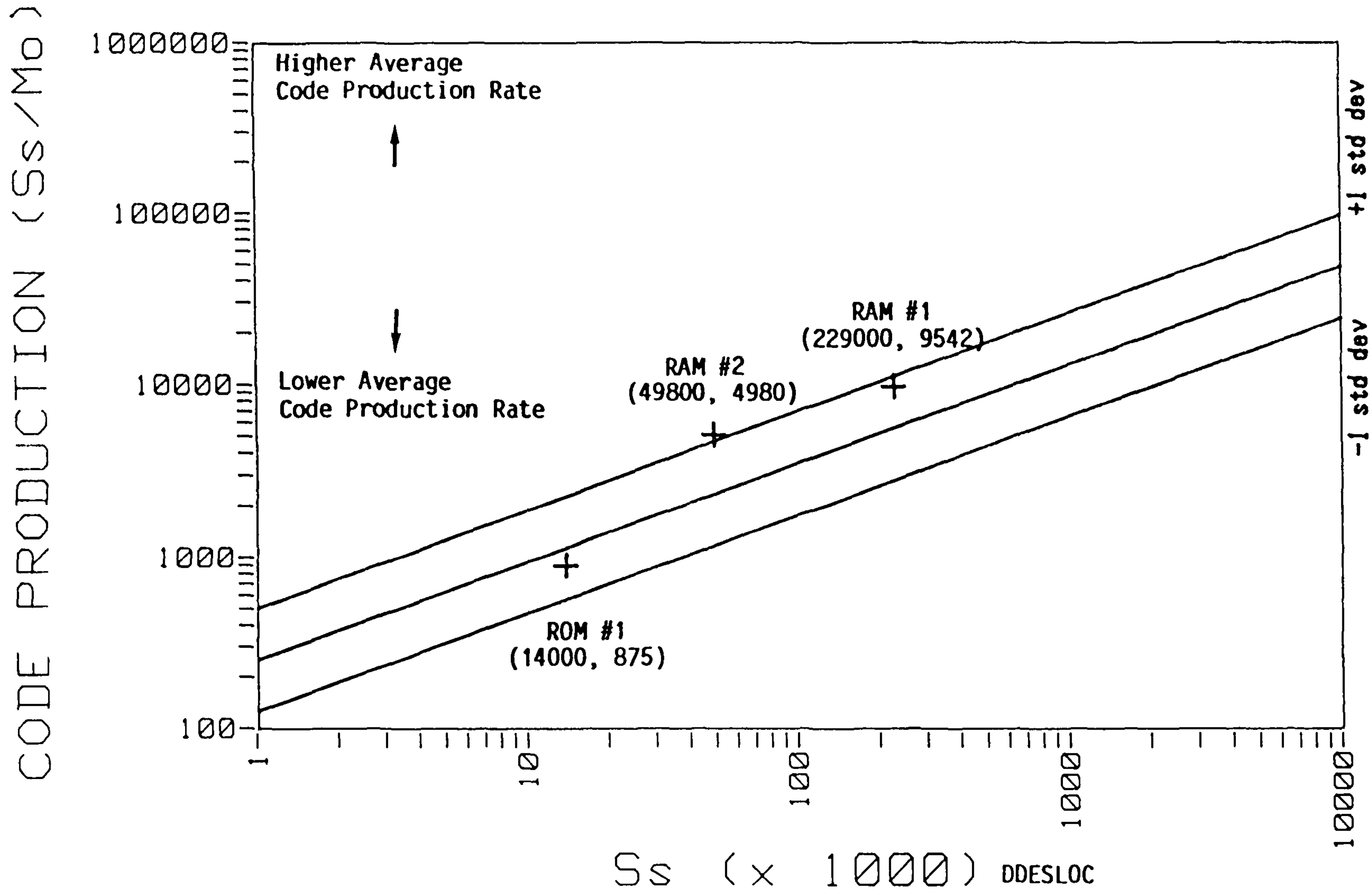


FIGURE 7

THE VIEWGRAPH MATERIALS

for the

L. PUTNAM PRESENTATION FOLLOW

MEASURING THE PROFICIENCY AND THE STYLE OF SOFTWARE DEVELOPERS

Lawrence H. Putnam
Quantitative Software Management, Inc.
1057 Waverley Way
McLean, Virginia 22101
(703) 790-0055

EVALUATION MEASURES TO DETERMINE REAL PRODUCTIVITY IN SOFTWARE DEVELOPMENT.

DEVELOPMENT ENVIRONMENT MEASURE

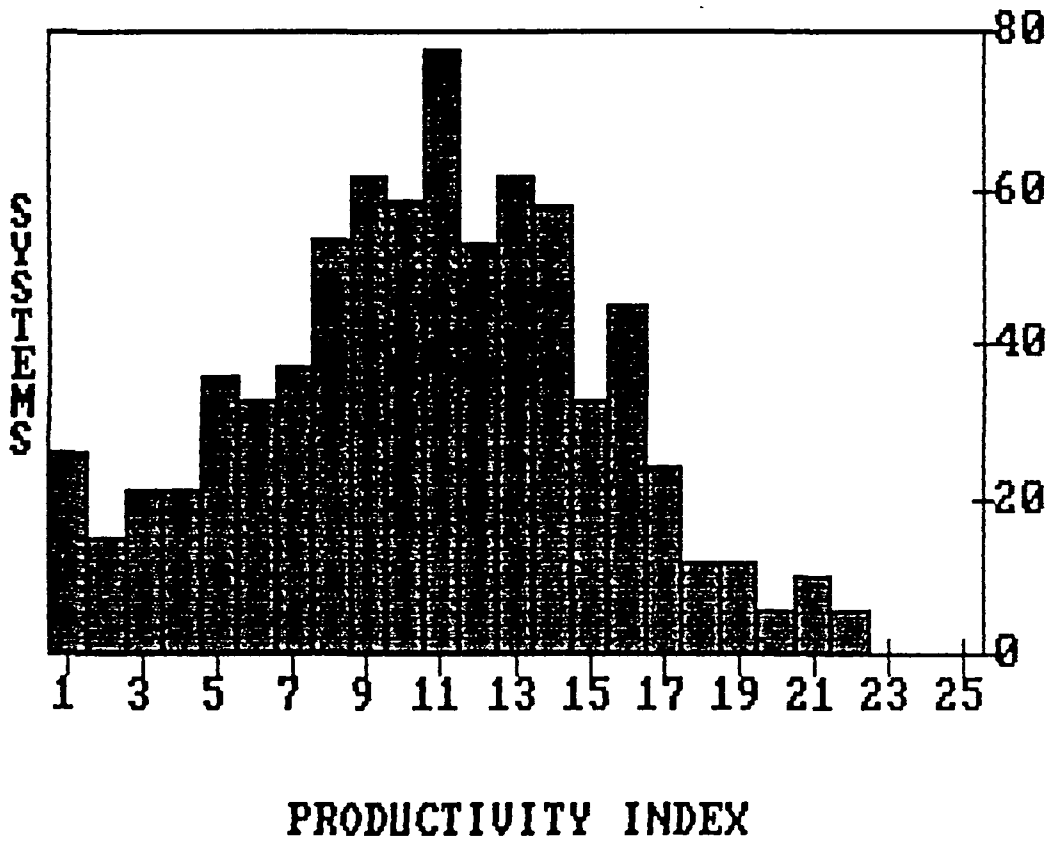
INCLUDES:

Management
Methodologies
Techniques
Computer based aids
Experience
Machine service
Type of application

SIMPLE SCALE

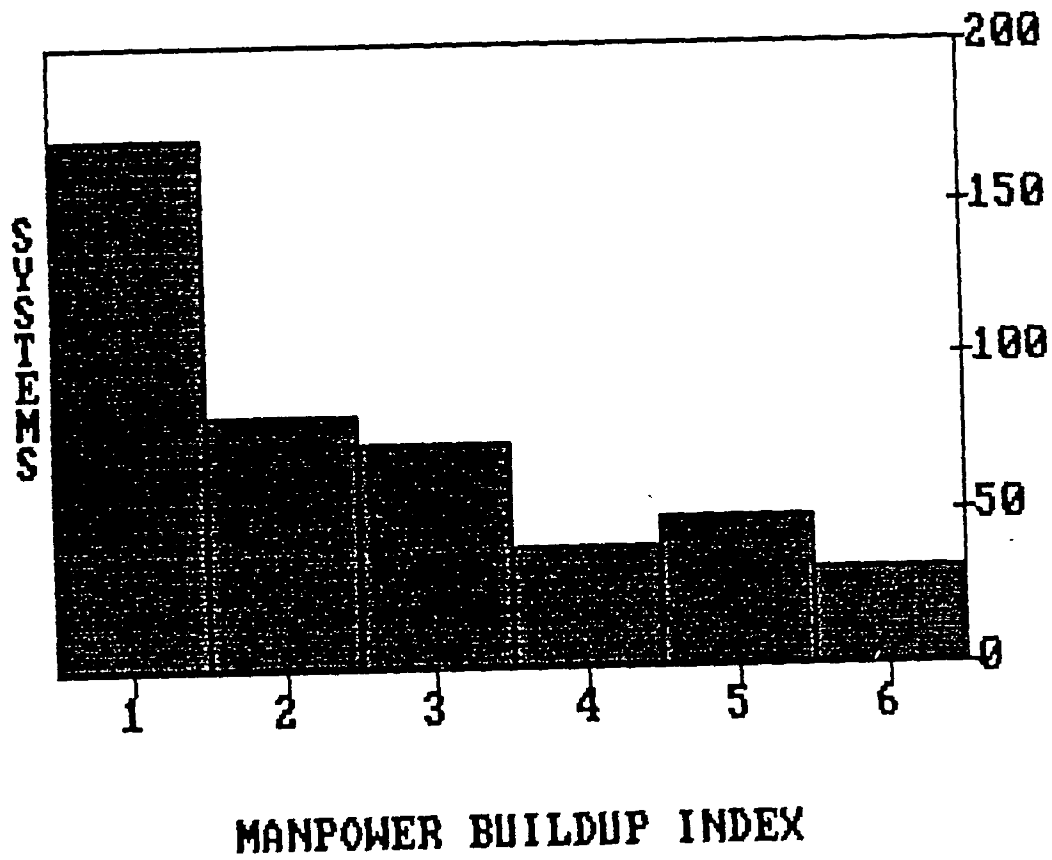
1, 2, 3,.......11, 12,....18, 19, 20, 21

Special systems	Telecom, Systems Software	Advanced Commercial


QSM SOFTWARE DATA BASE
(AS OF JAN 84)
SYSTEMS
80
60
40
20
0
1 3 5 7 9 11 13 15 17 19 21 23 25
PRODUCTIVITY INDEX

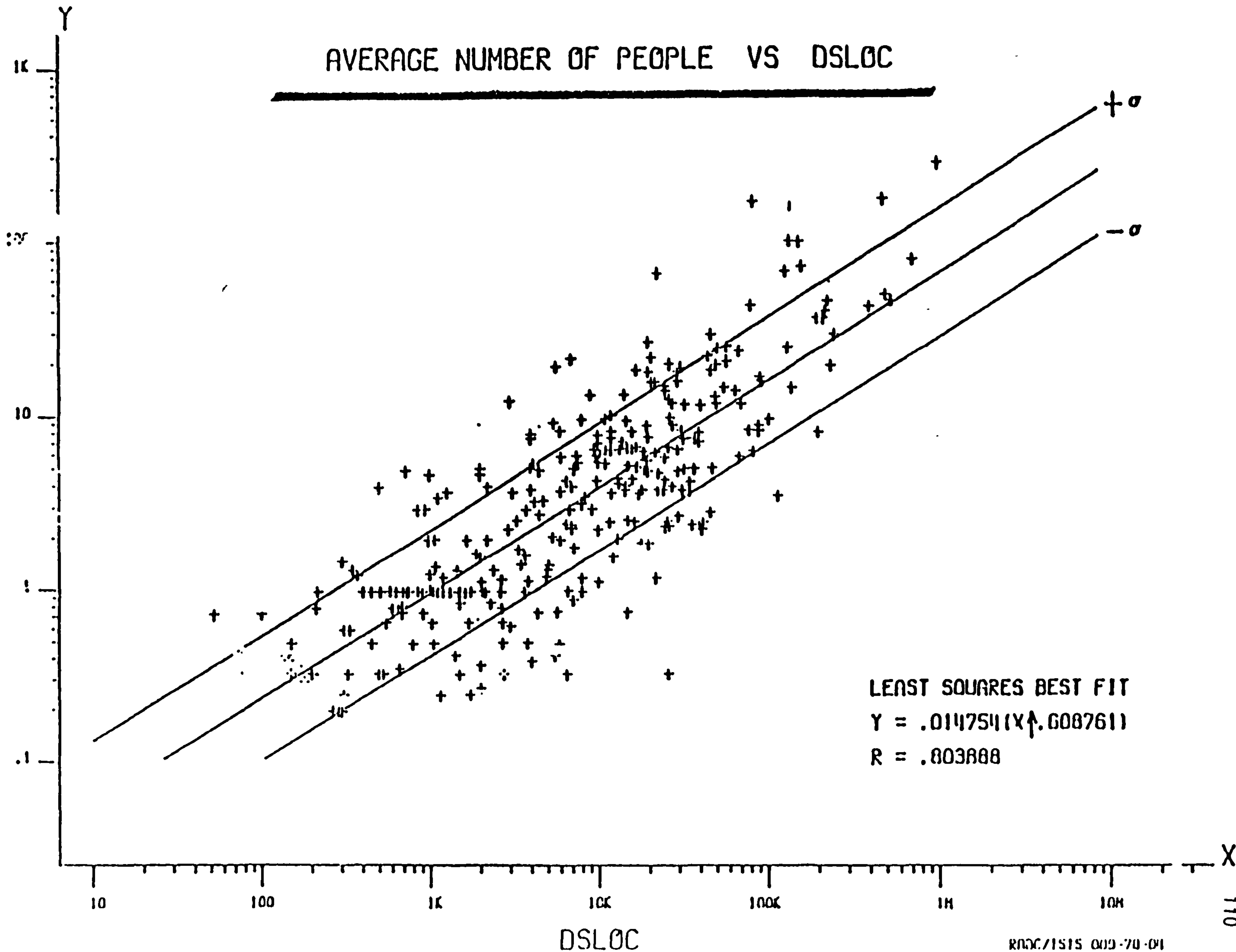
A MEASURE OF STYLE -- THE MANPOWER BUILDUP INDEX

A SIMPLE SCALE	BUILD UP RATE	PROBLEM TYPE
1	Slow	Sequential All new design
2	Mod. slow	Mod. Seq. Mostly new design
3	Moderate	Mod. parallel Some new design
4	Rapid	Parallel Little new design
5	Very rapid	Very parallel Almost no new design
6	Extremely rapid	Totally parallel No new design


QSM SOFTWARE DATA BASE
(AS OF JAN 84)
SYSTEMS
200
150
100
50
0
1
2
3
4
5
6
MANPOWER BUILDUP INDEX

SOME DATA–BASED MEASURES

* AVERAGE MANPOWER vs. SIZE
* EFFORT vs. SIZE
* DURATION vs. SIZE
* AVERAGE CODE PRODUCTION RATE vs. SIZE
* AVERAGE PRODUCTIVITY vs. SIZE

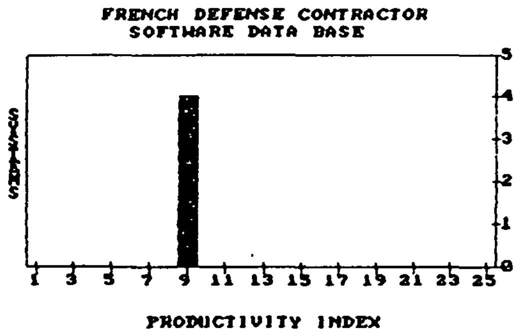

AVERAGE NUMBER OF PEOPLE VS DSLOC
AVERAGE NUMBER OF PEOPLE (TMM/TM)
Y
1K
10
1
.1
X
10
100
1K
10K
100K
1M
10M
DSLOC
+σ
−σ
LEAST SQUARES BEST FIT
Y = .014754(X↑.608761)
R = .803888

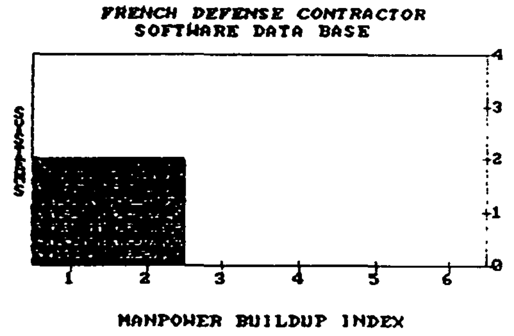
CALIBRATE INPUT SUMMARY

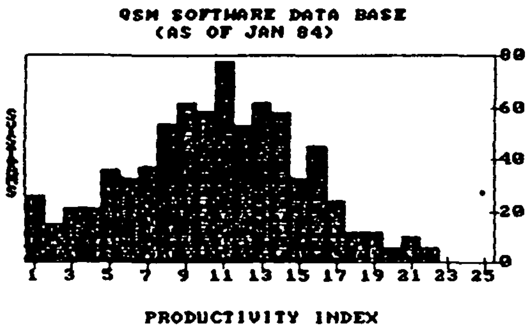
SYSTEM NAME	SIZE (SS)	TIME (MOS)	EFFORT (MM)	APPLICATION TYPE	OPERATIONAL DATE
CIMSA4	10840	19.0	38	REAL TIME	0383
CIMSA5	11000	12.0	18	COMMAND AND CONTROL	0383
CIMSA6	12294	12.0	22	COMMAND AND CONTROL	0383
CIMSA7	13000	12.0	31	COMMAND AND CONTROL	0383
CIMSA8	17024	22.0	34	REAL TIME	0383
CIMSA9	36900	24.0	76	COMMAND AND CONTROL	0383

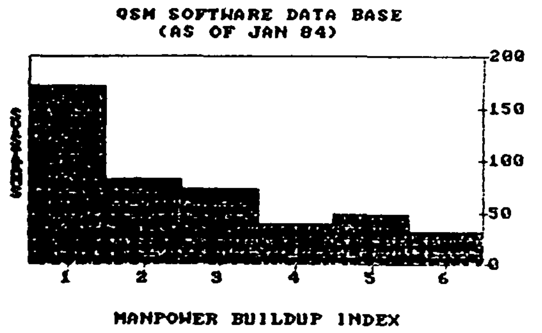
MANAGEMENT METRICS

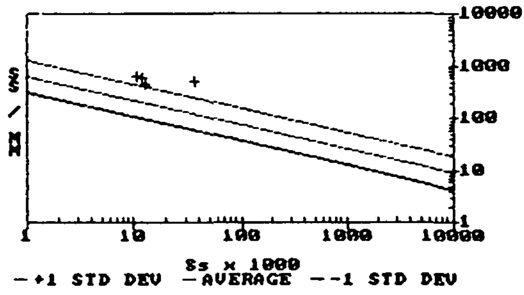
SYSTEM NAME	PRODUCTIVITY INDEX	MANPOWER BUILDUP INDEX	PRODUCTIVITY (SS/MM)	AVG MANPOWER (MM/MO)	AVG CODE PRODUCTION RATE (SS/MO)
CIMSA4	5	1	285	2	571
CIMSA5	9	1	611	2	917
CIMSA6	9	2	559	2	1025
CIMSA7	9	2	419	3	1083
CIMSA8	7	1	501	2	774
CIMSA9	9	1	486	3	1538

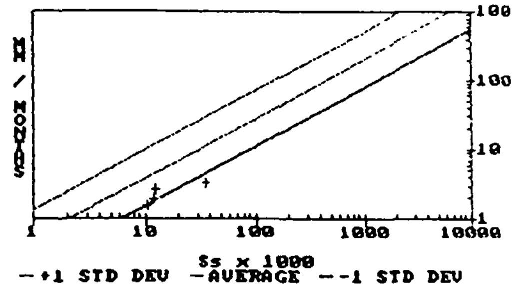

FRENCH DEFENSE CONTRACTOR
SOFTWARE DATA BASE
SYSTEMS
0
1
2
3
4
5
1 3 5 7 9 11 13 15 17 19 21 23 25
PRODUCTIVITY INDEX

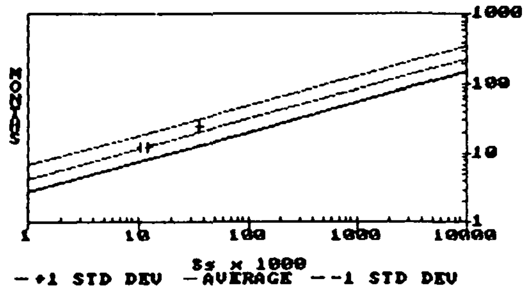

FRENCH DEFENSE CONTRACTOR
SOFTWARE DATA BASE
SYSTEMS
0
1
2
3
4
1 2 3 4 5 6
MANPOWER BUILDUP INDEX

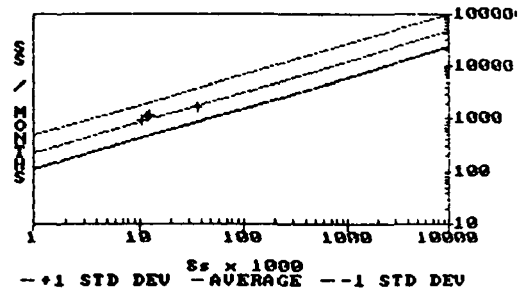

QSM SOFTWARE DATA BASE
(AS OF JAN 84)
SYSTEMS
0
20
40
60
80
1 3 5 7 9 11 13 15 17 19 21 23 25
PRODUCTIVITY INDEX

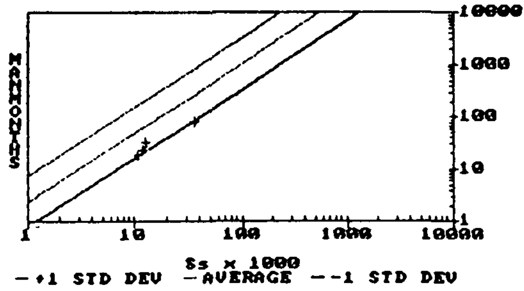

QSM SOFTWARE DATA BASE
(AS OF JAN 84)
SYSTEMS
0
50
100
150
200
1 2 3 4 5 6
MANPOWER BUILDUP INDEX


PRODUCTIVITY
Command & Control Systems
SS / MM
1
10
100
1000
10000
Ss x 1000
— +1 STD DEV — AVERAGE — -1 STD DEV


AVERAGE MANPOWER
Command & Control Systems
MM / MONTHS
1
10
100
1000
10000
Ss x 1000
— +1 STD DEV — AVERAGE — -1 STD DEV


DURATION
Command & Control Systems
MONTHS
1
10
100
1000
10000
Ss x 1000
— +1 STD DEV — AVERAGE — -1 STD DEV


AVG CODE PRODUCTION RATE
Command & Control Systems
SS / MONTHS
1
10
100
1000
10000
100000
Ss x 1000
— +1 STD DEV — AVERAGE — -1 STD DEV

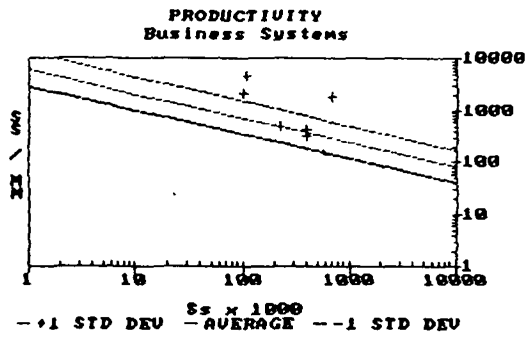

EFFORT
Command & Control Systems
MANMONTHS
1
10
100
1000
10000
Ss x 1000
— +1 STD DEV — AVERAGE — -1 STD DEV

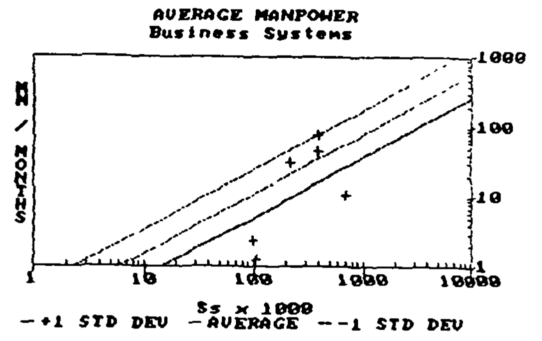
CALIBRATE INPUT SUMMARY

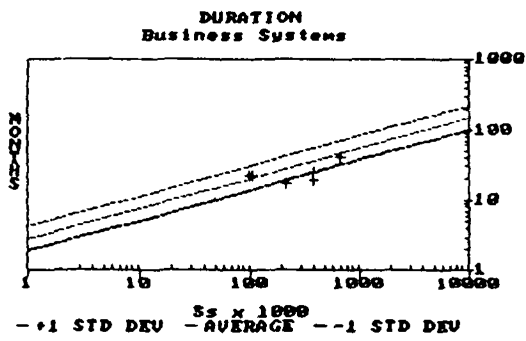
SYSTEM NAME	SIZE (SS)	TIME (MOS)	EFFORT (MM)	APPLICATION TYPE	OPERATIONAL DATE
JAPAN VENDOR 1	390000	24.0	1003	BUSINESS	0879
JAPAN VENDOR 1	224000	16.0	472	BUSINESS	0881
JAPAN VENDOR 2	400000	18.0	1324	BUSINESS	0381
PARTS NUMBER	108000	21.0	25	BUSINESS	0182
RFM	100000	21.0	48	BUSINESS	1082
MATERIALS MGMT	700000	38.0	384	BUSINESS	0183

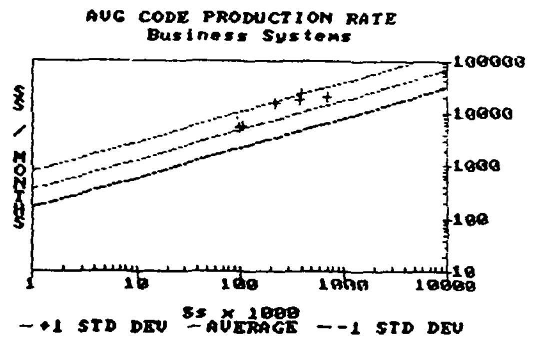
MANAGEMENT METRICS

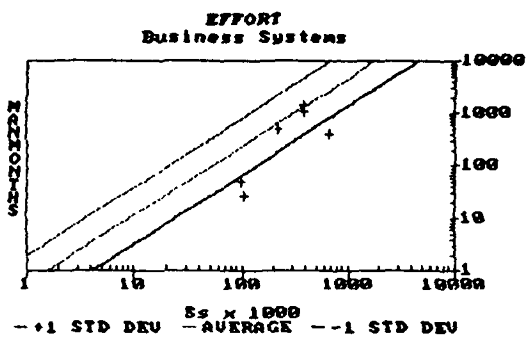
SYSTEM NAME	PRODUCTIVITY INDEX	MANPOWER BUILDUP INDEX	PRODUCTIVITY (SS/MM)	AVG MANPOWER (MM/MO)	AVG CODE PRODUCTION RATE (SS/MO)
JAPAN VENDOR 1	16	3	389	42	16250
JAPAN VENDOR 1	17	3	475	30	14000
JAPAN VENDOR 2	17	5	302	74	22222
PARTS NUMBER	16	1	4320	1	5143
RFM	15	1	2083	2	4762
MATERIALS MGMT	17	1	1823	10	18421

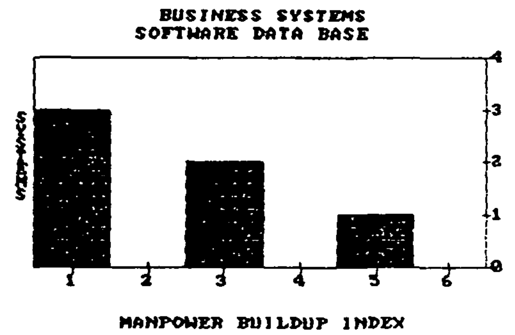

PRODUCTIVITY
Business Systems
SS / MM
Ss x 1000
—+1 STD DEV —AVERAGE —-1 STD DEV

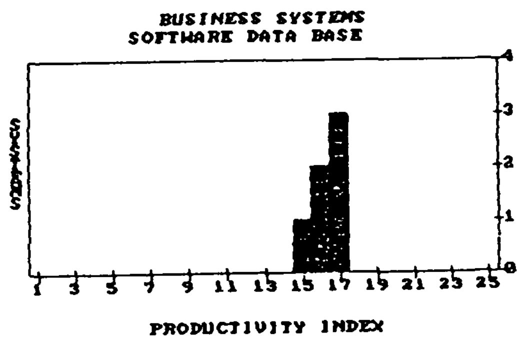

AVERAGE MANPOWER
Business Systems
MM / MONTHS
Ss x 1000
—+1 STD DEV —AVERAGE —-1 STD DEV

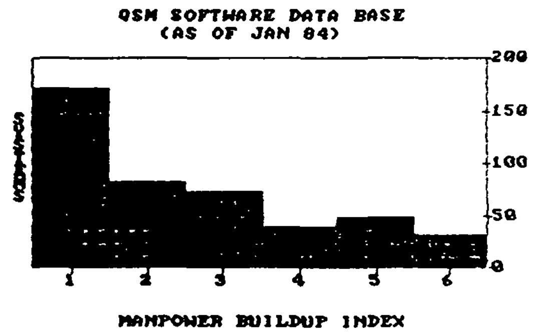

DURATION
Business Systems
MONTHS
Ss x 1000
—+1 STD DEV —AVERAGE —-1 STD DEV

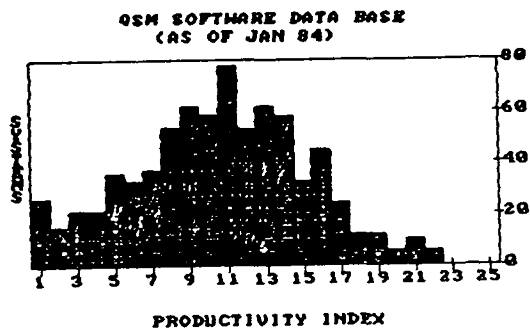

AVG CODE PRODUCTION RATE
Business Systems
SS / MONTHS
Ss x 1000
—+1 STD DEV —AVERAGE —-1 STD DEV


EFFORT
Business Systems
MANMONTHS
Ss x 1000
—+1 STD DEV —AVERAGE —-1 STD DEV


BUSINESS SYSTEMS
SOFTWARE DATA BASE
SYSTEMS
4
3
2
1
0
1 2 3 4 5 6
MANPOWER BUILDUP INDEX


BUSINESS SYSTEMS
SOFTWARE DATA BASE
SYSTEMS
4
3
2
1
0
1 3 5 7 9 11 13 15 17 19 21 23 25
PRODUCTIVITY INDEX


QSM SOFTWARE DATA BASE
(AS OF JAN 84)
SYSTEMS
200
150
100
50
0
1 2 3 4 5 6
MANPOWER BUILDUP INDEX

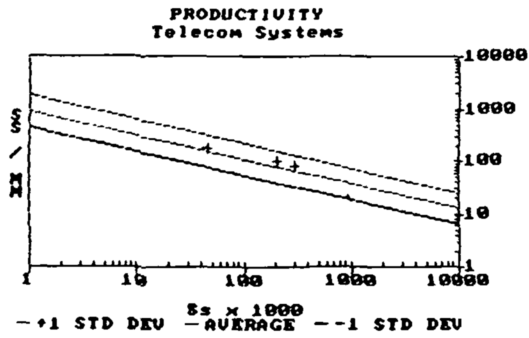

QSM SOFTWARE DATA BASE
(AS OF JAN 84)
SYSTEMS
80
60
40
20
0
1 3 5 7 9 11 13 15 17 19 21 23 25
PRODUCTIVITY INDEX

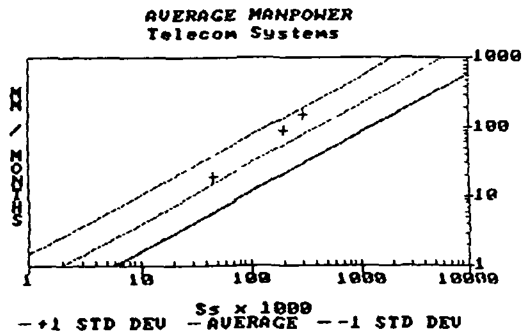
CALIBRATE INPUT SUMMARY

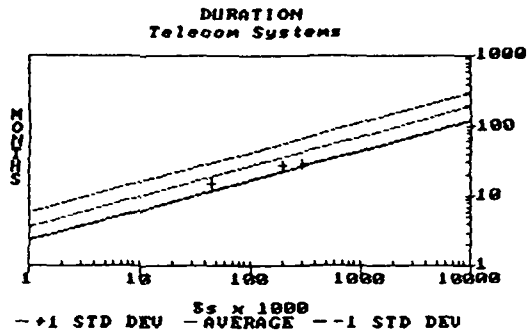
SYSTEM NAME	SIZE (SS)	TIME (MOS)	EFFORT (MM)	APPLICATION TYPE	OPERATIONAL DATE
DIGITAL SWITCH	46900	15.0	270	TELECOM&MSG SWITCH	0283
D700 SWITCH	210000	26.0	2185	TELECOM&MSG SWITCH	0883
US SWITCH	308000	27.0	3860	TELECOM&MSG SWITCH	0682

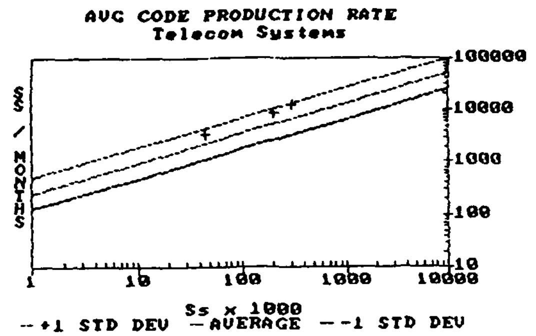
MANAGEMENT METRICS

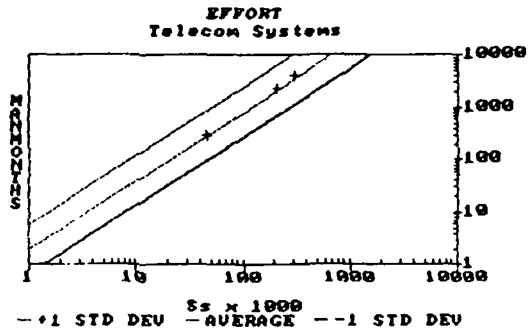
SYSTEM NAME	PRODUCTIVITY INDEX	MANPOWER BUILDUP INDEX	PRODUCTIVITY (SS/MM)	AVG MANPOWER (MM/MO)	AVG CODE PRODUCTION RATE (SS/MO)
DIGITAL SWITCH	11	3	174	18	3127
D700 SWITCH	12	4	96	84	8077
US SWITCH	12	4	80	143	11407

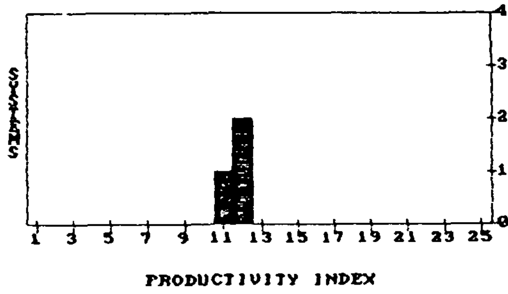

PRODUCTIVITY
Telecom Systems
SS / MM
10000
1000
100
10
1
1
10
100
1000
10000
Ss x 1000
—+1 STD DEV —AVERAGE —-1 STD DEV

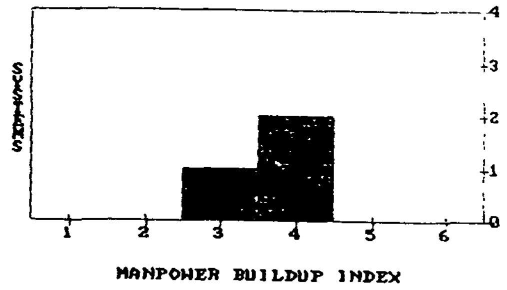

AVERAGE MANPOWER
Telecom Systems
MM / MONTHS
1000
100
10
1
1
10
100
1000
10000
Ss x 1000
—+1 STD DEV —AVERAGE —-1 STD DEV

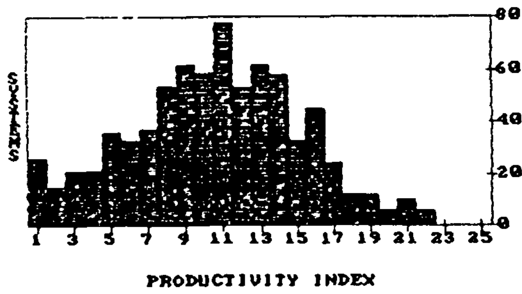

DURATION
Telecom Systems
MONTHS
1000
100
10
1
1
10
100
1000
10000
Ss x 1000
—+1 STD DEV —AVERAGE —-1 STD DEV

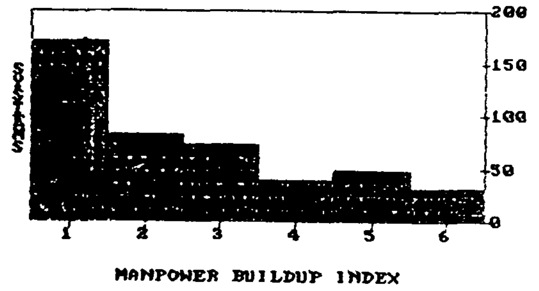

AVG CODE PRODUCTION RATE
Telecom Systems
SS / MONTHS
100000
10000
1000
100
10
1
10
100
1000
10000
Ss x 1000
—+1 STD DEV —AVERAGE —-1 STD DEV


EFFORT
Telecom Systems
MANMONTHS
10000
1000
100
10
1
1
10
100
1000
10000
Ss x 1000
—+1 STD DEV —AVERAGE —-1 STD DEV


TELECOM SYSTEMS
SOFTWARE DATA BASE
SYSTEMS
4
3
2
1
0
1 3 5 7 9 11 13 15 17 19 21 23 25
PRODUCTIVITY INDEX


TELECOM SYSTEMS
SOFTWARE DATA BASE
SYSTEMS
4
3
2
1
0
1 2 3 4 5 6
MANPOWER BUILDUP INDEX


QSM SOFTWARE DATA BASE
(AS OF JAN 84)
SYSTEMS
80
60
40
20
0
1 3 5 7 9 11 13 15 17 19 21 23 25
PRODUCTIVITY INDEX


QSM SOFTWARE DATA BASE
(AS OF JAN 84)
SYSTEMS
200
150
100
50
0
1 2 3 4 5 6
MANPOWER BUILDUP INDEX

IT IS POSSIBLE TO MEASURE REAL PRODUCTIVITY IN SOFTWARE

- THE MEASURES ARE:

  * AVERAGE MANPOWER vs. SIZE
  * EFFORT vs. SIZE
  * DURATION vs. SIZE
  * AVERAGE CODE PRODUCTION RATE vs. SIZE
  * AVERAGE PRODUCTIVITY vs. SIZE
  * SOFTWARE EFFICIENCY INDEX
  * MANPOWER BUILDUP INDEX

- TAKEN TOGETHER THEY TELL A CONSISTENT STORY.

- COMPARED WITH A STRATIFIED DATA BASE THEY TELL HOW EFFECTIVE THE DEVELOPER IS.

MANAGEMENT IMPLICATIONS

* The Productivity Index is a good overall measure of efficiency. It is determined from size, time and effort; therefore, it is a good measure of a real productivity gain. It can be used to measure improvements over time. In ideal situations where additional project information is available it can isolate tools, methodologies or management practices that had a high payoff.

* The Manpower Buildup Index is a good measure of staffing style.

* SCHEDULE and STAFFING are determined and controlled by management. So management can have a big impact on "effective productivity" in software development. This means that staffing decisions effect the BOTTOM LINE.

Quantitative Software Management, Inc.

A BASIC MANAGEMENT TENET IS:

"If you can't measure it, you can't manage it."

Richard L. Nolan

Managing the Crisis
in Data Processing
HARVARD BUSINESS REVIEW
March–April 1979

"If you do things the way you have always done them, you will get what you have always gotten before."

Buick

Scientific American, May 1984

OMIT D13

PROGRESS IN REFINING AND USING A CONSISTENT SET OF SOFTWARE PRODUCTIVITY MEASURES

(C)Copyright by Quantitative Software Management, Inc.
April 1984

by

Lawrence H. Putnam
Douglas T. Putnam
Lauren P. Thayer

In the mid 1970's Lawrence Putnam developed a equation that explained the behavior of software systems. He called it the software equation. It is written in the form:

$$Ss = Ck \times K^{1/3} \times Td^{4/3}, \text{ Subject to } K/Td^3 \leq G$$

Notice that there are only four terms in the basic equation. The components are defined as:

- Ss - The total number of DDESLOC **
- Ck - An overall efficiency-complexity measure
- K - Total Life-cycle effort
- Td - The development time
- G - maximum manpower acceleration possible for a class of system

Thus, a given product can be developed in Td amount of time, for Y amount of effort, at Z efficiency level.

The software equation can be thought of as a powerful trade-off law. A given product developed in a fixed environment, could be developed with many different time-effort combinations, all of which would satisfy the equation. However, because of the time and effort exponents, the equation gives dramatic results. With these exponents small changes in time produce substantial changes in effort. In practical use the software equation has demostrated it can be a high leverage software management function.

Over the past 5 years we have analysed data from over 2000 software projects. Our intention was to independently validate the software equation. Of those 2000 projects some 803 had complete data and have been entered into our database. With this data we have been able to prove that the exponents are very close to the true behavior.

** DDESLOC is the notation use for Delivered, Developed, Executable, Source Lines of Code.

In late 1982 we established regression trend lines for our database. Regression lines were developed for the measures listed below.

Productivity (Ss/MM) vs DDESLOC
Schedule vs DDESLOC
Effort vs DDESLOC
Average Manpower (MM/MOS) vs DDESLOC

In the initial analysis we observed clusterings in the values of Ck. The cluster patterns were related to application type. It was thought that the trend lines might be correlated to Ck. Each application type should have it's own family of trend lines that would shift up or down according to the range of Ck values present.

By mid 1983 the database was large enough to stratify according to application type. The major categories identifed were:

Real time Embedded systems
Avionics systems
Management Information systems
Scientific systems
Command and Control systems
Systems software
Microcode and Firmware systems

The curve fitting exercise confirmed our thoughts. The trend lines did shift. Micro-code and firmware were located at the low end of the spectrum. This software had low values for Ck, low productivity, took a long time, was quite expensive and demanded more people relative to similar sized projects. The MIS application were at the high end of the spectrum. These systems had high values for Ck, high productivity, shorter schedules, were less expensive and used fewer people relative to their size.

Variability around the average trend lines was still a concern. Could the software equation explain that variability? There is a ratio that effectively measures the application of effort over time. This measure is called the Manpower Buildup Gradient. It is defined as K/Td^3. It discloses the style of the software development organization. High values (generally larger than 20) are present when parallel effort is possible and management is willing to commit what ever resources are necessary to get a system built fast. Low values are more typical of sequential efforts (design intensive processes) or a management constrainted situation (limited available manpower).

New data was analyzed using the new trend lines as a basis for comparison. We found that it told a consistent and unambigious story. The typical behavior pattern for systems with a steep manpower buildup rate is: modest schedule compression, lower productivity (Ss/MM), higher average code production (Ss/Mos), requiring more effort and more people. Conversely, systems with a gradual manpower buildup rate had slightly longer schedules, much higher productivity, lower average code

production rates, requiring less effort and fewer people.

CASE STUDY (A Major Computer Vendor)

An independent data set from a major U.S. computer manufacturer illustrates these points. The systems used in this analysis come from a manufacturing facility dedicated to building smart IBM compatible mainframe terminals. The software that drives the most recent family of terminals is written primarily in C language with a small portion of assembly code. The primary system functions are diagnostics, memory management, and communication. This family of products has a limited market share. The costs associated with product development are high but can be recovered along with a profit if the manufacturer can deliver the product within a narrow market window. The software is the guts of the product and therefore critically important. Company management is willing to dedicate large software development staffs to get whatever schedule compression is needed to meet the market demands (regardless of whether the schedule is realistic or not).

The data from three systems developed recently at this plant is summarized in the top portion of Table 1. Notice that two systems are RAM based. Due to a hardware constraint the third system had to be written so that it could reside in ROM. The unique problems present on the ROM development include severely limited memory and very high performance specifications. High quality was essential on the ROM system because it involved a manufacturing process and would be costly to replace once it was in the field.

The bottom portion of Table 1 summarizes important calculations made on the input data. The column titled Productivity Index uses a linear sequence of numbers that relate to the actual Ck values in parenthesis. Likewise the Manpower Buildup Index corresponds to the Manpower Buildup Gradient values in parenthesis. Notice that there is a big difference between the Ck values of the RAM and ROM based systems.

The Manpower Buildup Gradient for all three systems are high. The value calculated from RAM #2 is more than double that of RAM #1. According to the software trade-off law there should be a noticable difference between the two systems for the time and effort required to complete these projects. The other measures summarized in Table 1 are dependent on system size. Taken out of this context they are not meaningful. However, if we compare them against a baseline for their own size and application then they will be meaningful.

GRAPHICAL ANALYSIS (QSM System Software Database)

Figure 1 is a frequency graph of the Manpower Buildup Index for the three systems. Two observations can be made from this chart. The development style of this company is to staff up quickly and use alot

of people. This management practice can be explained by the market environment. A second observation is worthy of notice. The RAM based systems have different manpower buildup index measures. The data from RAM #1 calculates a 3. RAM #2 calculates a 4. The management trade-off decisions that produced these systems should show exchanges of time and effort according to the software equation.

Figure 2 show the distribution of Ck. The graph utilizes an index which the Ck values fall within. It is immediately obvious that there is a big difference between the two types of software implementations. The difference can not be attributed to the function that the software performs. They are quite similar. Rather, it is in the way the code has to be designed and written for the particular implementation which is quite different. The ROM software is more difficult because it requires designing tricky code overlays to meet memory restriction and needs constant performance tuning. It must be bug free before it is burned into ROM. RAM #2 has a higher Ck than RAM #1. With a higher Ck RAM #2 utilized less overall effort. The higher efficency of RAM #2 will counteract the nonlinear effort increase attributed to the steeper Manpower Buildup Rate that RAM #2 had.

In Figure 3, the data is superimposed on the average manpower trend lines for the System software database. Notice that the scales are logrithmic. The log scales turn the non-linear trends into straight lines. The absissa (X axis) represents the total number of Delivered Developed Executable Source Lines of Code. The ordinate (Y axis) is the average number of people (MM/Td). There are three trend lines drawn on the graph. The middle line is the best regression fit for all the data contained in the Systems software database. The high and low lines are the plus and minus one standard deviation bounds. Each cross represents the calculated average manpower plotted at the reported size.

The ROM system required significantly more people than other comparably sized systems software projects. On the other hand, the RAM based systems are very economic in their use of manpower compared to the industry average for their size. In a relative sense RAM #1 has a lower manpower utilization compared to RAM #2. This can be attributed to the more gradual manpower buildup rate.

Figure 4 is a similar portrayal of the database. In this case we will compare Total Manmonths against the system size. Since manmonths are proportional to cost, this graph compares these systems for cost effectiveness. The ROM system is significantly more expensive. The RAM systems are well below the industry average. RAM #1 appears to be a little less expensive compared to RAM #2.

Figure 5 compares the data against the productivity trend lines. The ROM system is close to two standard deviations lower than the average for that sized project. The RAM based systems again are better than the industry average. Notice that RAM #1 has a better relative postion compared to it's size.

Figure 6 starts to disclose the trade-off situation. This figure compares the data against the trend lines for average project

duration. It is no surprise that the ROM system took significantly longer than the average. The RAM systems are interesting. RAM #2 is somewhat shorter than the average. In constrast RAM #1 is a little longer than the average. The pattern seems to coming together. RAM #2 as you recall had the steeper manpower buildup rate. The objective must have been to get the system built fast. The system was built in a shorter time but in a relative sense it required a lot more effort and people. The difference would be more pronounced if the values of Ck were the same. The non-linearities present in software equation are still powerful enough counteract the lower efficency of RAM #1.

The pattern continues in Figure 7. The ROM system is again below the industry average. The Ck for the ROM system was well below the Systems software average and explains why it compares in such a unfavorable way. RAM #2 experienced a rapid manpower buildup and therefore had a higher code production rate. RAM #1 had a more gradual manpower buildup and a lower relative code production rate.

CONCLUSION

The trend lines presented in this paper can be useful in a number of ways. They provide a baseline of comparision from which software developers can compare their performance against a large database of similar projects. This will often identify a organizational style. In this case study it was possible to quantify the organizational style using the Manpower Buildup Gradient. Additionally we were able to show that the developer was a better than average producer on RAM based systems. The Ck associated with the ROM system suggests that it is a different class of work. When this system is compared against the Firmware database it is very creditable.

It is important to recognize that there are non-linearities present in the software process. The non-linearities are tied to system size. For comparative purposes we must always make judgements based on similar sizes. In the past the tendency has been to calculate a few ratios on several projects and then compare them without any regard to amount of functonality that was created. This practice can be very misleading and dangerous. The method describe in this paper used in a thoughtful analytic manner can be very helpful.

There are some problems associated with curve fitting that should be pointed out. With the non-linearities present in software, small data sets will often produce wide variations in slope. Any effort (MM) dependent ratios are particularly problematic. Productivity has consistently proven to be the most sensitive. Of all fits on productivity (Ss/MM) that we have made we have never been able to get a r squared value better than (.02). The nonlinearities in the terms productivity is composed of are responsible for this. To work around this situation we have chosen to combine a theoretical slope tuned by the actual data.

It is possible to extend this approach. The present plans include providing for a reliability comparision. Right now the error database is not large enough to get totally reliable statistics but before too long we hope to establish those trend lines as well. The database will be analyzed to determine the improvement that is being made in each of the application areas over time. Some preliminary work in this area has been done and it looks very promising.

CALIBRATE INPUT SUMMARY

SYSTEM NAME	SIZE (SS)	TIME (MOS)	EFFORT (MM)	APPLICATION TYPE	OPERATIONAL DATE
ROM SOFTWARE #1	229000	24.0	869	SYSTEM SOFTWARE	
RAM SOFTWARE #2	49800	10.0	133	SYSTEM SOFTWARE	
ROM SOFTWARE #1	14000	16.0	178	SYSTEM SOFTWARE	

MANAGEMENT METRICS

SYSTEM NAME	PRODUCTIVITY INDEX	MANPOWER BUILDUP INDEX	PRODUCTIVITY (SS/MM)	PEAK MANPOWER (PEOPLE)	AVG CODE PRODUCTION RATE (SS/MO)
RAM SOFTWARE #1	12 **(15,976)**	3 **(23)**	264	56	9542
RAM SOFTWARE #2	14 **(20,348)**	4 **(53)**	374	19	4980
ROM SOFTWARE #1	5 **(2,100)**	3 **(39)**	79	17	875

Table 1

COMPUTER VENDOR
STAFFING BUILDUP DEV DATABASE

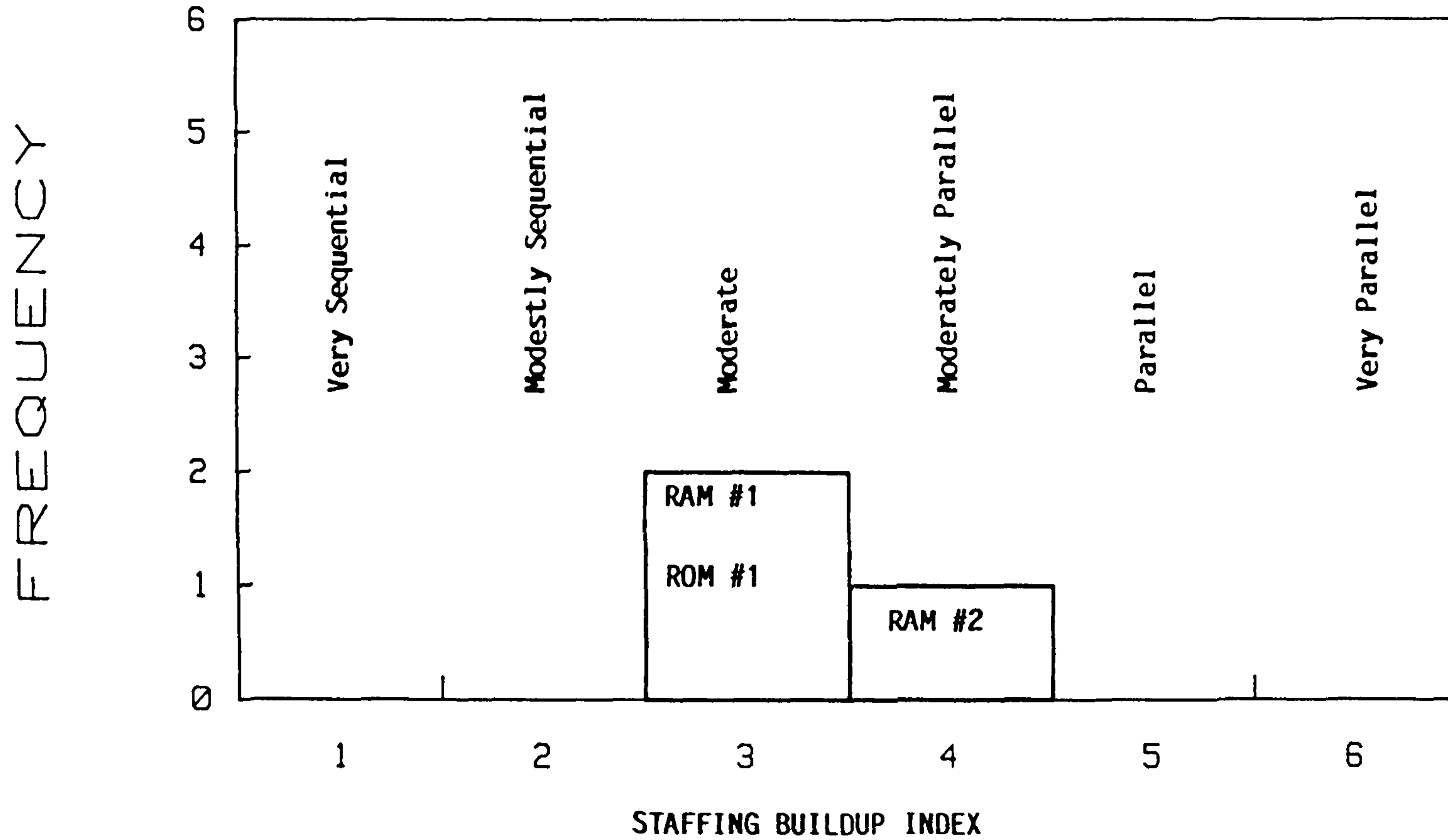


FIGURE 1

COMPUTER VENDOR
TECHNOLOGY FACTOR DEV DATABASE

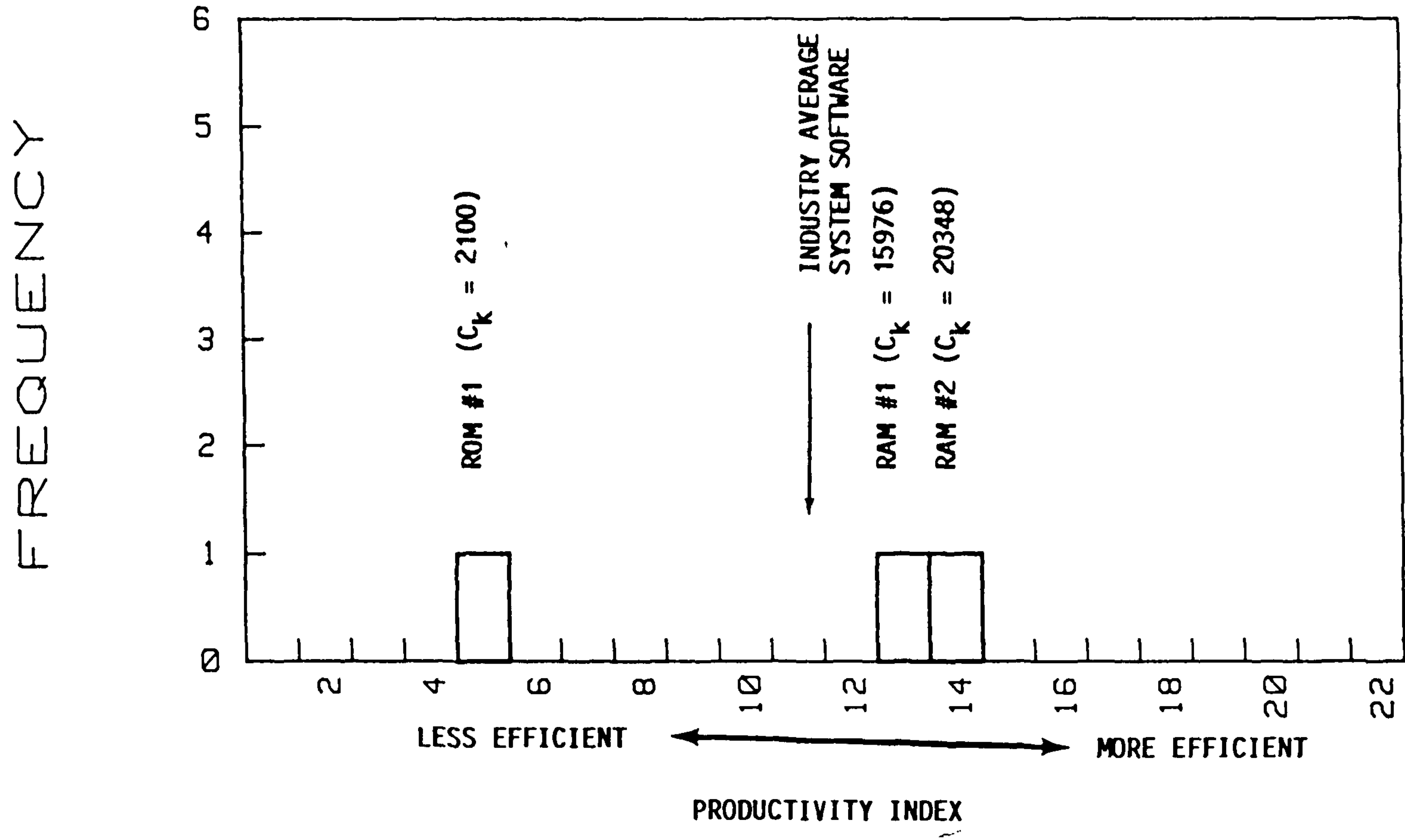


FIGURE 2

SYSTEM SFTW - AVERAGE # OF PEOPLE
VS. Ss

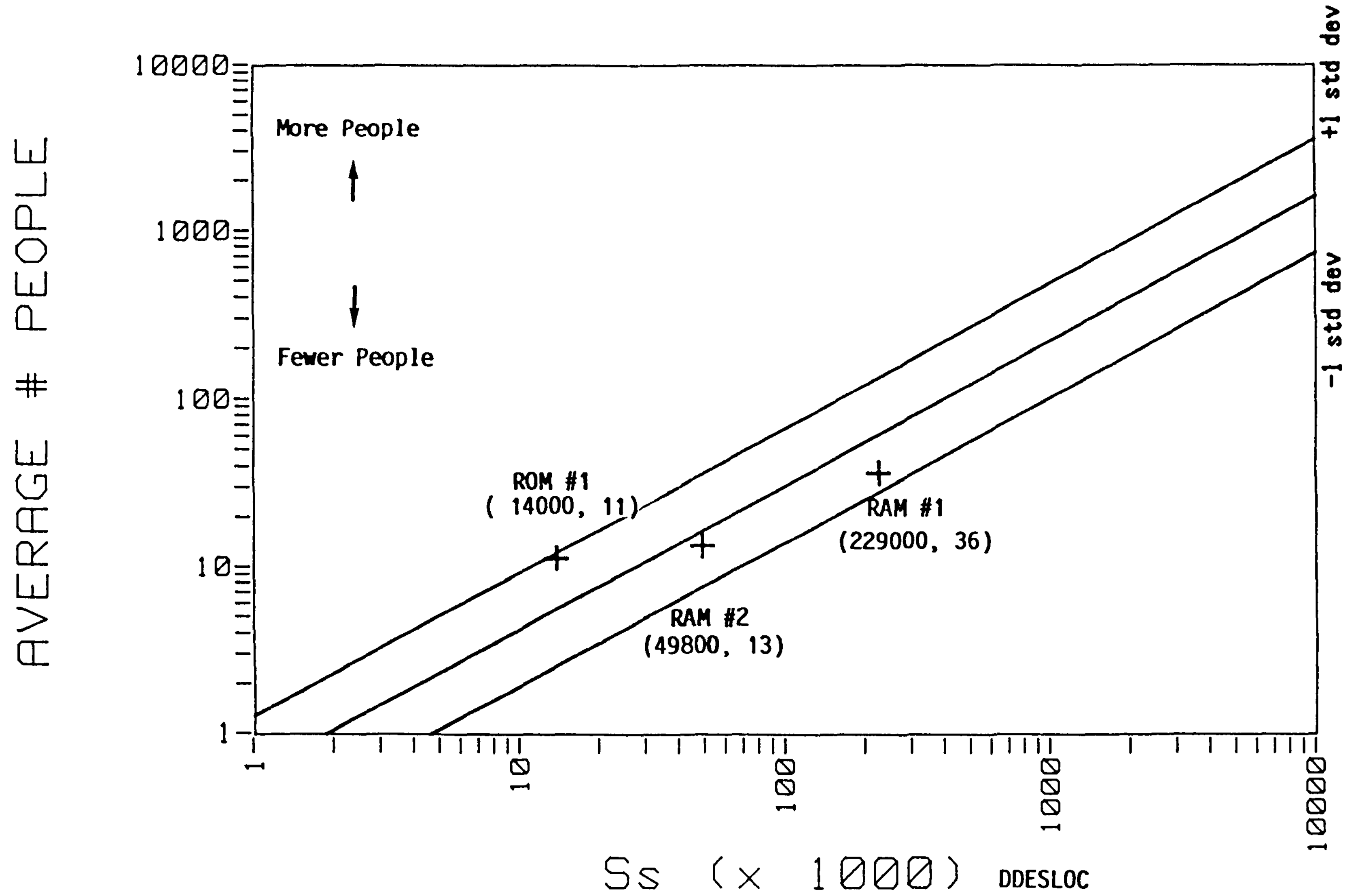


FIGURE 3

SYSTEM SFTW - TOTAL MANMONTHS VS. Ss

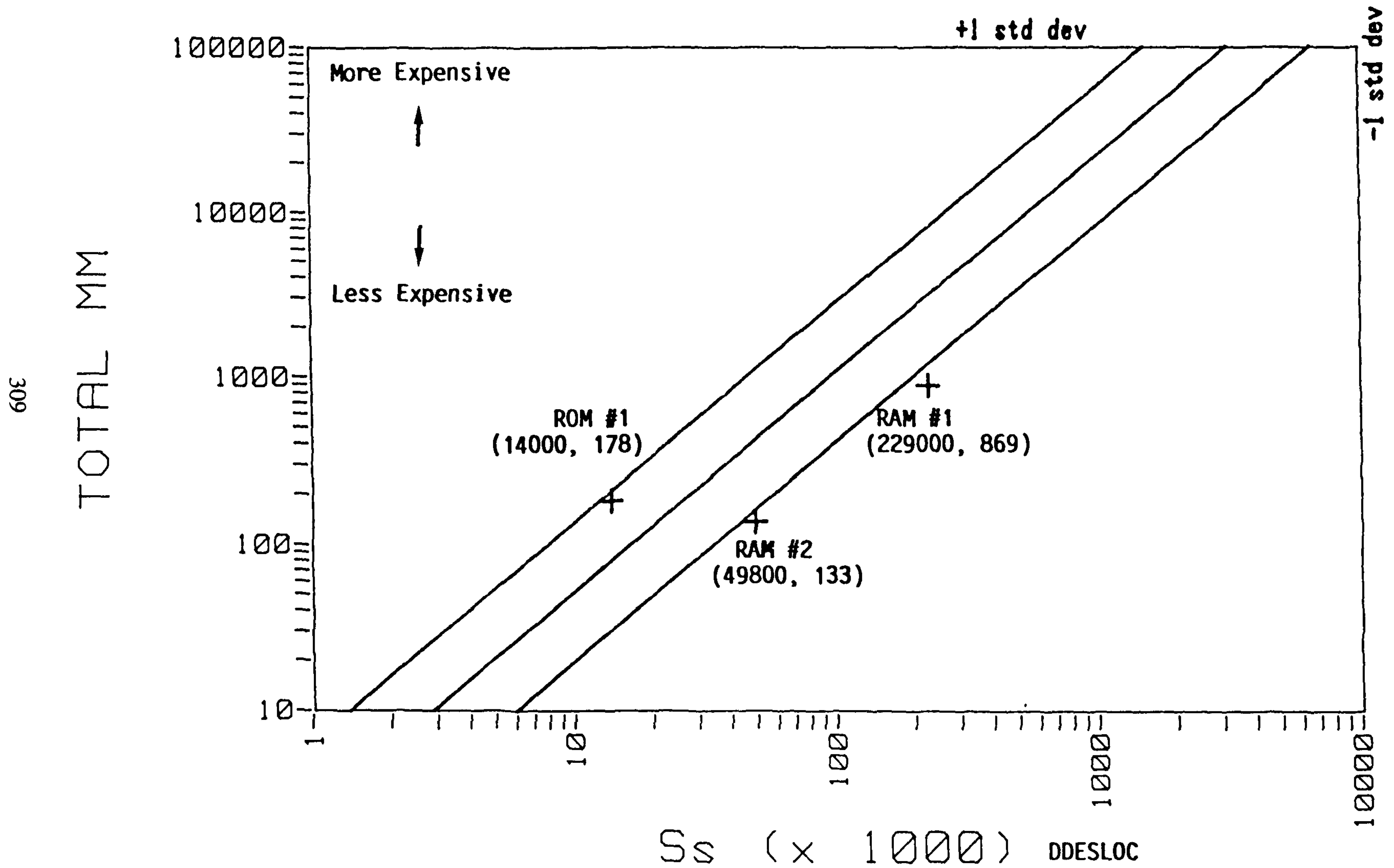


FIGURE 4

SYSTEM SFTW - PRODUCTIVITY VS. Ss

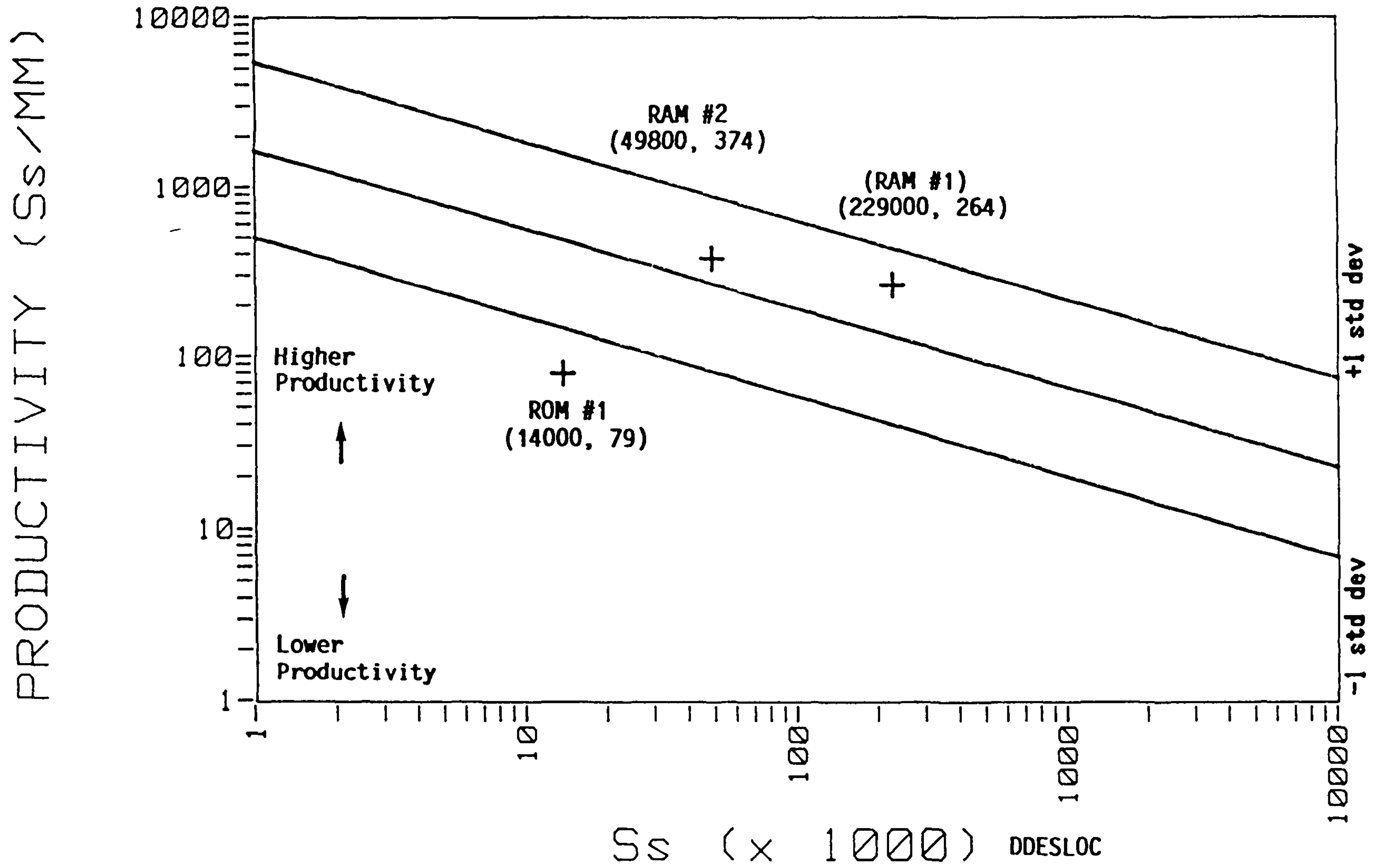


FIGURE 5

SYSTEM SFTW - PROJECT DURATION (Mos)
VS. Ss

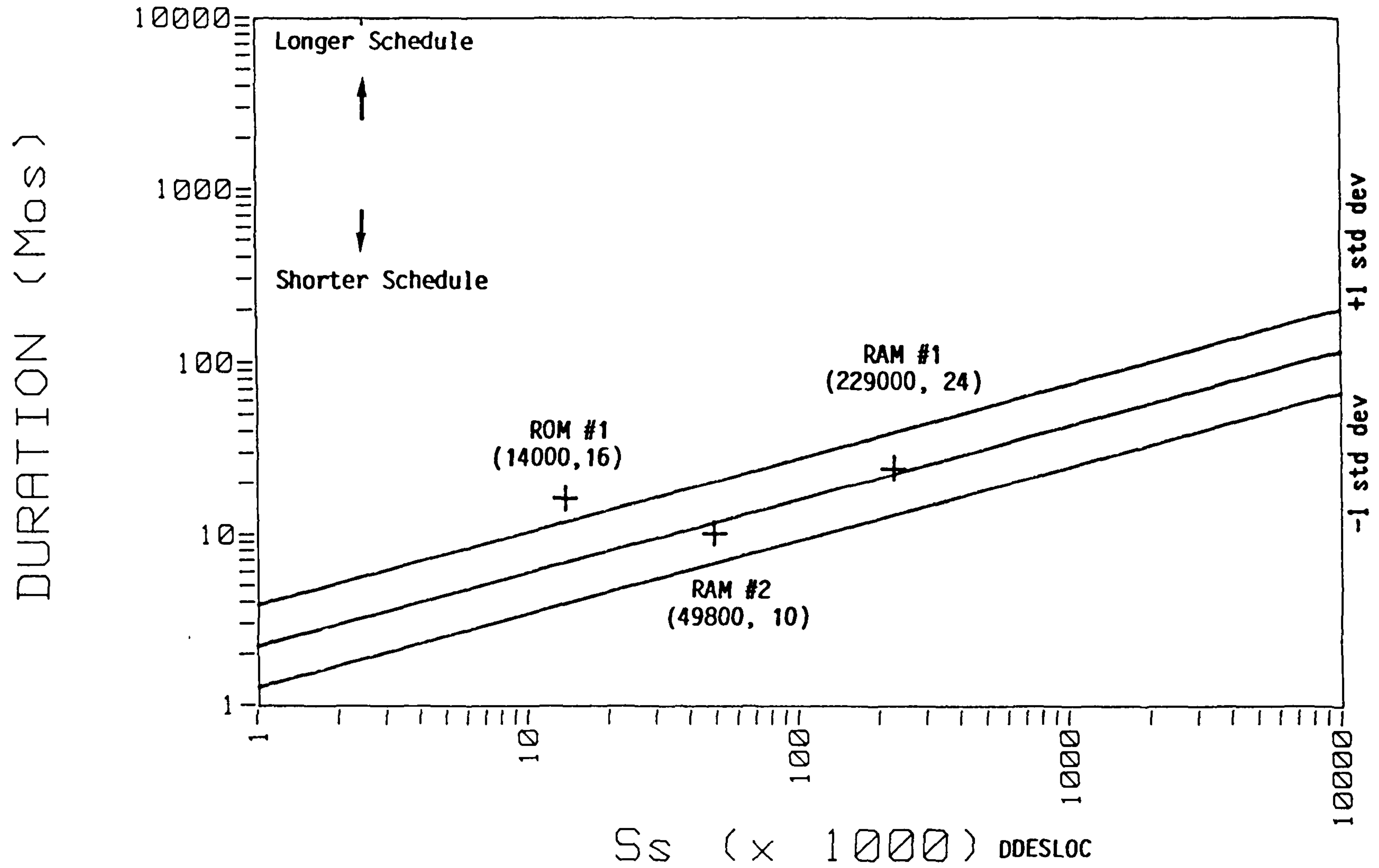


FIGURE 6

SYSTEM SFTW - CODE PRODUCTION VS. Ss

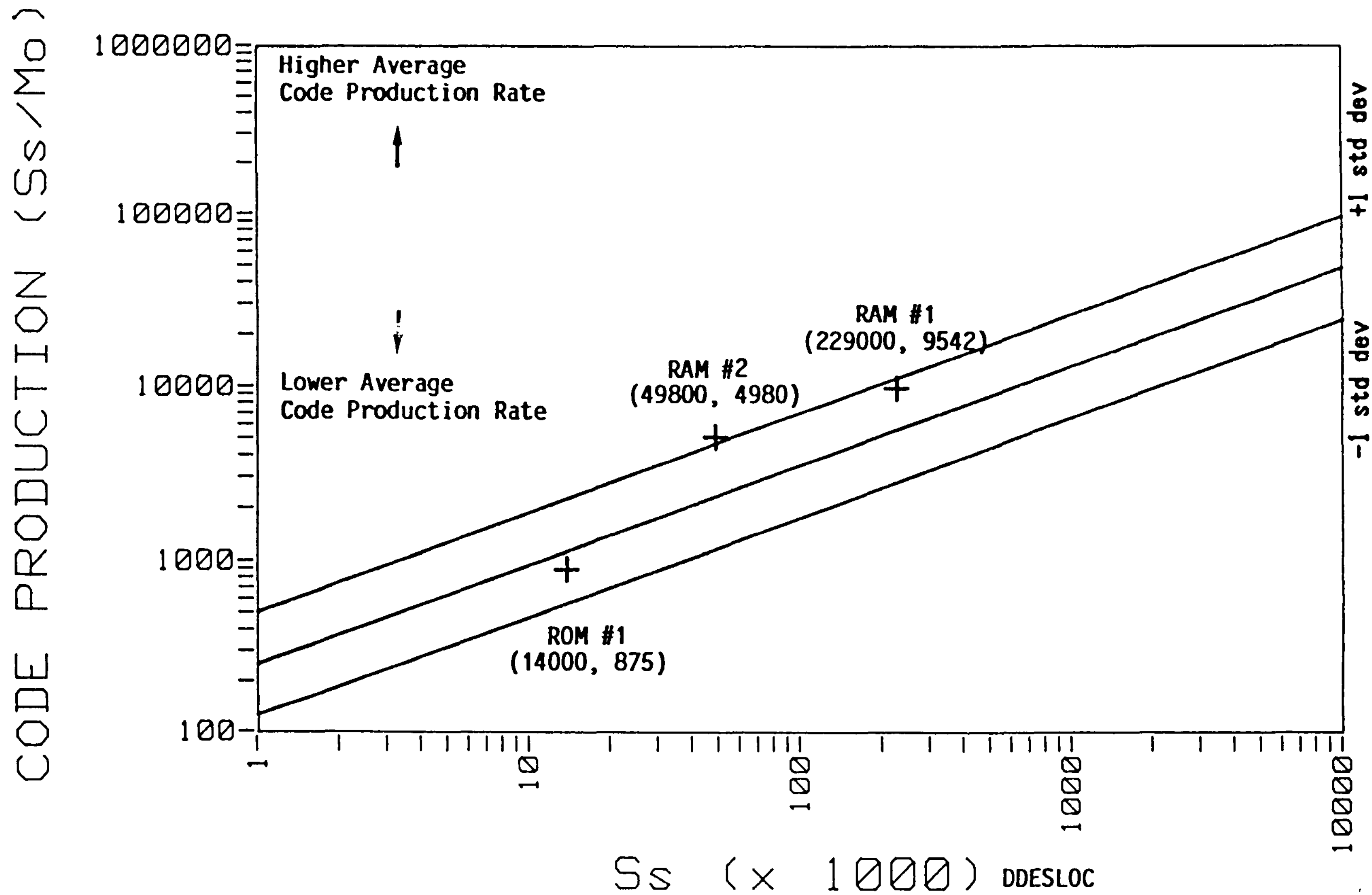


FIGURE 7

N86-19980 D14

TAILORING A SOFTWARE PRODUCTION ENVIRONMENT FOR A LARGE PROJECT

(Abstract)

David R. Levine

Intermetrics, Inc.
733 Concord Ave.
Cambridge, Mass 02174
617: 661-1840

A software production environment was constructed to meet the specific goals of a a particular large programming project. This paper will discuss these goals, the specific solutions as implemented, and our experiences on a project of over 100,000 lines of source code.

The base development environment for this project was an ordinary PWB Unix (tm) system. Several important aspects of the development process required support not available in the existing tool set (e.g. SCCS, make).

Version management:

Many systems provide source library tools with version numbering and similar support. We wanted to track the version number of a module at all stages of the development process: within the source libraries; as source and object in private development directories; and as constituents in both private and official load modules. A method was developed to automatically maintain the version identification of each module in a form as to be easily visible and checkable by standard tools, in particular by the linker.

In addition, the space / time balance of the source library required evaluation. We desired fast access to the library, and did not anticipate the need for reference to very old versions. Furthermore, any number of standard tools would be applied to the text, including both unix tools such as grep, sed, and nroff, and other of our own devising. A library was developed in which the text was held in clear text, thus providing both simplicity and speed in processing. Simple file system techniques provided version and access control.

Separate Compilation:

The development language supported separate compilation, but with a caveat emptor attitude towards interface consistency. We required a more rigorous system to maintain correctness and control recompilation; to avoid version skew and yet minimize unnecessary recompilation. The project was based on a decentralized methodology, in which every module had the responsibility of defining its own interface. A system was developed in which the interface definitions were provided in the same files as the functions they described, and then extracted for inclusion by other units. Techniques similar to those used for basic version control provided firm checking (including linker error reports) on version skew.

Incremental Development:

Our development model is one of continuous integration. At any point, the developer must see a stable, official baseline configuration, plus some personally constructed set of modules being modified. Standardized handling was desired to facilitate sharing of experimental modules among different developers, and to ease the transition into new configurations. Uniform procedure would allow automatic logging of activity, desirable for management purposes, to allow us to pick up if a key person were absent, and to help automate the "gate" (configuration acceptance) cycle. The system as developed relied on the version visibility scheme to allow private modules to coexist in public areas, and to even obviate the absolute need for recompilation of a module when submitted to the gate.

Experience:

This environment was implemented on Unix, originally as a collection of shell scripts. It served to support development by up to 20 programmers, on a large, highly interconnected program. Over a period of two years, over 200 gate cycles were run, as the program grew to over 700 modules and over 100,000 lines of source code.

Reflection has shown both strengths and weaknesses of the approach. For instance, a project of this size seems to require less strong interconnection, and less changeable interfaces; that has major implications for the support system. More recent systems and tools, such as RCS on Unix and Apollo's DSEE system, offer better solutions to the basic space/time tradeoff in the source library.

TAILORING A SOFTWARE PRODUCTION ENVIRONMENT FOR A LARGE PROJECT

David R. Levine

Intermetrics, Inc.
733 Concord Ave.
Cambridge, Mass. 02138

Overview

This paper describes a software production environment that we developed to support a large compiler project. The host environment was a Unix * system, with a Remote Job Entry link to a large batch mainframe. The project size reached 100K lines of code in over 700 source modules, with approximately two dozen developers at peak strength.

Figure 1 lists some of the problems involved in the design of this environment. One of the factors unique to this project was the particular choice of methodology and implementation language, which mandated a high level of supplementary support from the environment. The methodology in question includes a heavy reliance on data abstractions, which tends to lead to a highly modular design. We intended to maintain this design structure in implementation as well. In a language like Ada **, which is designed to support this methodology, such a strategy presents little difficulty. In our case, the implementation language could be teased into supporting the design structure, but at the expense of a great deal of potential complexity. It was clear at the outset that additional tool support was needed to provide the kind of consistency and configuration management necessary to keep the methodology from getting in the way. A specific penalty of complexity is a greater need to assure correctness, especially in regard to separate compilation; in a system with many components, one needs more positive measures.

For a large project, we knew a good configuration management system was necessary. We intended to use an incremental development strategy, in which one gets the core of the system working and then glues on more and more functionality. This strategy places a severe strain on configuration management, imposing simultaneously the requirements of development and maintenance phases. Firm configuration management is required to

* Unix is a registered trademark of Bell Laboratories.

** Ada is a registered trademark of the United States Government, Ada Joint Program Office.

Large Project

- USUAL WORRIES
- CONSERVE RESOURCES
- METHODOLOGY ISSUES

LOTS of COMPILATION UNITS and MINIMAL SUPPORT from LANGUAGE

⟹ GOOD MGMT OF SEPARATE COMPILATION (INTERFACES, RECOMPILATION)

Good Configuration Mgmt

- SOURCE CODE VERSION MGMT
- INCREMENTAL DEVELOPMENT

STABLE OPERATIONAL VERSION, FREQUENTLY UPDATED

Figure 1.

- ACCOUNTABILITY THRUOUT PROCESS, NOT JUST ON OFFICIAL VERSION
  - LESS CONFUSION
  - SAVE RESOURCES
  - STRONG CM SUPPORT AT DEVELOPER LEVEL
- AUTOMATIC OR SEMI-AUTOMATIC METHODS PREFERRED

  BUILD TOOLS

  AS REQUIRED
- DESIGNATED CONFIGURATION LIBRARIAN

Figure 2.

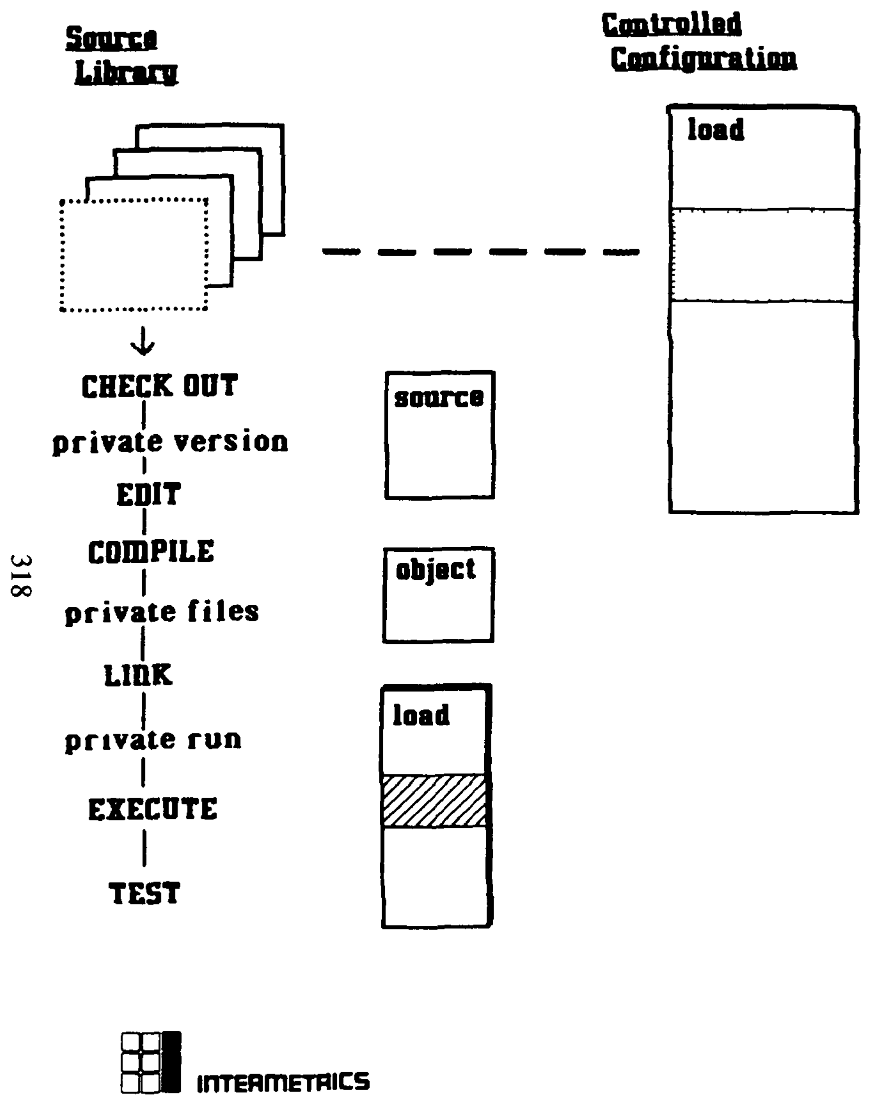


Figure 3.

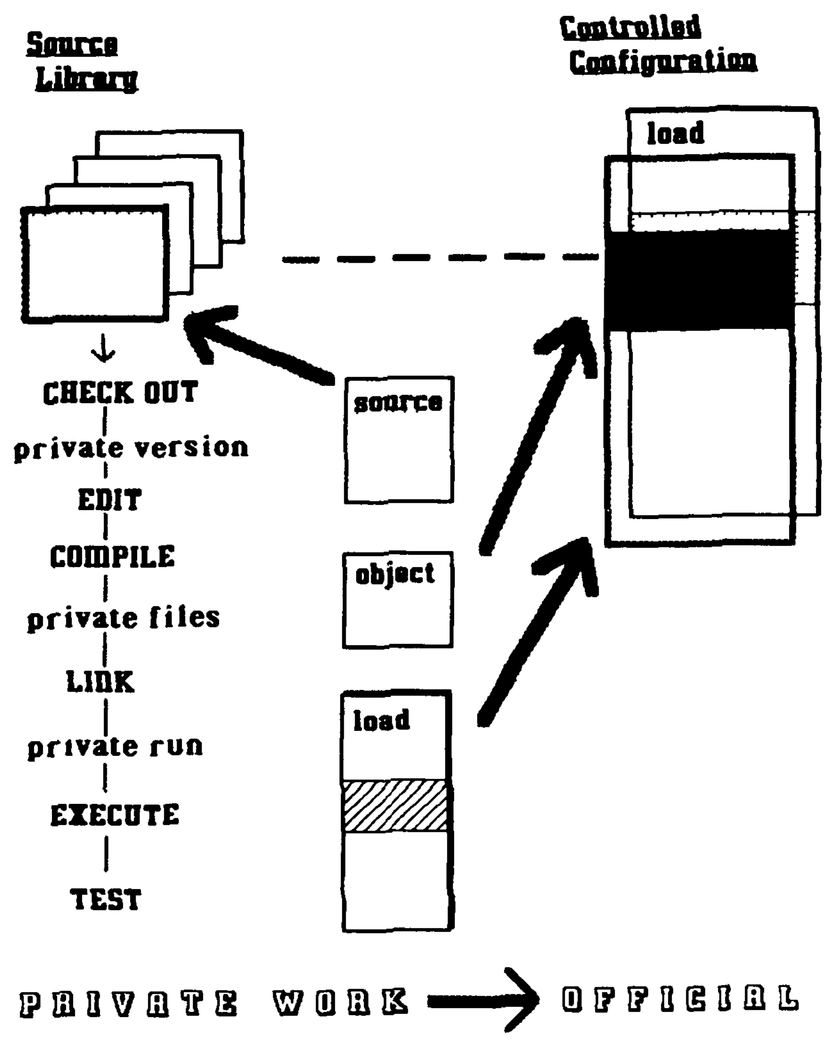


Figure 4.

maintain at all times a good, working version; at the same time, the flexibility required during full development must also be provided.

Figures 3 and 4 show some of the flow of activity in such an environment. One will note the flow of modules from the controlled [source] library into the private workspace of individual developers and then back again into the library to make up a new controlled configuration.

Traditional systems often enshrine a basic dichotomy between the strict control of the configuration managed world and the unstructured freedom of the individual developer. We felt there were numerous advantages available if we could successfully bridge this gap. We thus found ways to provide better support for the developers, to help them systematize their activities in a constructive manner by using the good features of the CM system. We were able to reduce operational confusion at the individual level. The similarity of procedures facilitated cooperation between developers. And the CM system benefits too, in reducing the complexity of its new-version acceptance phase.

Another of the major principles which guided us was that of accountability. We wanted to track the pieces of the system as they moved around in the development process. To the extent possible, we wanted self-identification of the various components and files in our system. We needed to create and maintain correct interfaces, and be able to verify that necessary recompilations had been done.

The Unix environment encourages the development and use of tools. The library and configuration management systems that we build saw us through over 9000 versions of the source modules, and 200 CM acceptance cycles over a two year period. We employed a Configuration Librarian to manage the centralized functions of creating a new configuration; other elements of the system were automatically handled by tools invoked by individual developers.

Positive Version Identification

Looking at a little more detail at the problem of module identification, we notice that there is a three-dimensional space to manage. The overall program under development consists of a great many individual modules. For each module, several versions may be in existence. And for each version, several forms must be handled: source, object, and as a component of the linked program. Figure 5 shows a simple example, with modules "A" and "B". Note the shaded version #23 of "A" which forms part of the current load module.

Proven source library systems exist, with good functionality. But their tracking and control is purely internal; once checked out, a file is essentially anonymous until validated either by check-in or acceptance into the

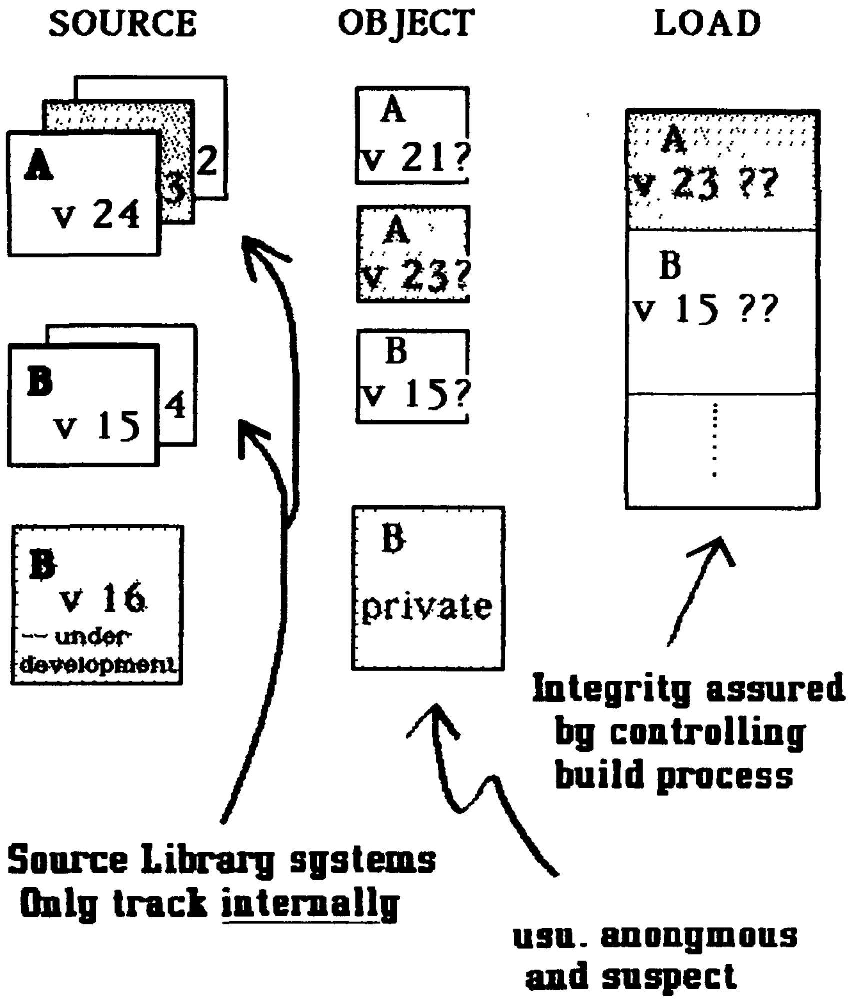


Figure 5.

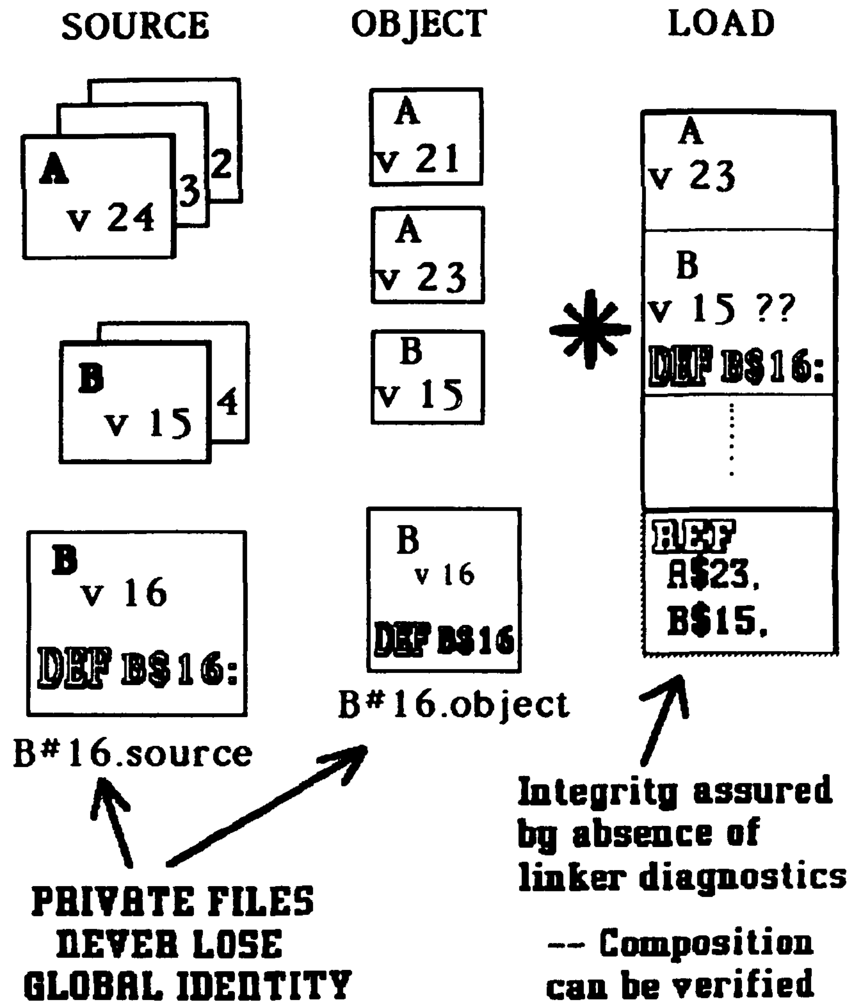


Figure 6.

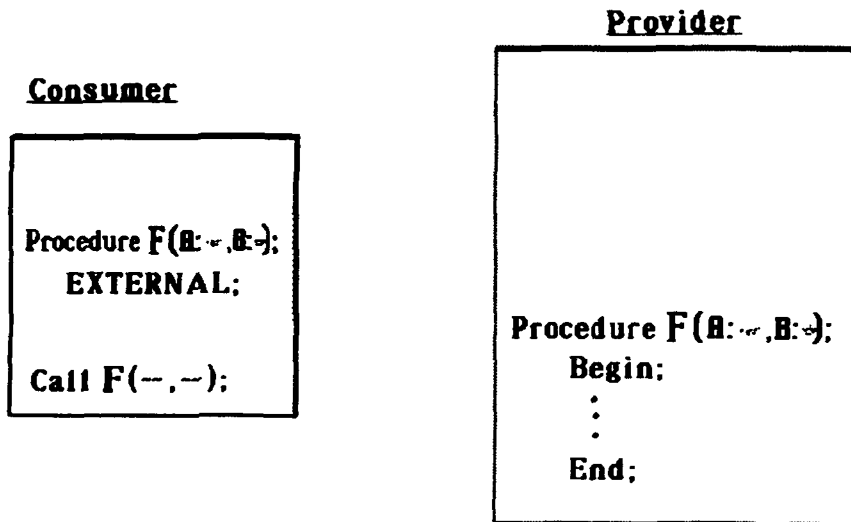
controlled configuration. Its identity is suspect; system integrity is assured by rigidly controlling the build process. (The individual developer typically exerts only modest control here, and thus is particularly susceptible to confusion.) Figure 5 shows a typical object module which claims to be version 23 of "A". How is that identity established? Solely by knowing how it was constructed. If any confusion sets in, the solution is to scratch everything and start over -- often an unacceptably expensive alternative.

Our system maintains order in two simple but effective ways. The first concerns file names. A private copy of a file -- "B" in the figure -- need not be anonymous. By checking it out, formally, the source library manager has reserved the next highest version number. We take the obvious step and assign that identity immediately, so that even in the private workspace the file carries its correct name and version number. (Cf. figure 6) Since that name and number combination is reserved, and unique throughout the development environment, even the "private" object files can be likewise tagged -- and furthermore be stored in a canonical file system, accessible to anyone who needs them.

The other aspect of maintaining order lies in a self-identification scheme, so that a module's identity can be firmly established independent of external artifacts such as file names. For this purpose, we create a special variable in each module. This variable is not part of the program logic; it is purely part of the CM system. The name of the variable is a concatenation of the module name and version number, automatically adjusted when a module is checked <u>out</u> of the source library. (Thus within module "B" is the variable "B$16"; see figure 6.) We use a variable, and not just a comment, so that the version identification shows up in the compiler output. Furthermore, the CM symbol is given <u>external</u> status, making it visible -- by name -- in the object module. In particular, as an external symbol it is processed by the linkage editor.

In addition, the configuration librarian maintains a separate source file with a set of complementary CM symbols. The "DEF" construct in the primary module source serves to provide a definition of the CM symbol in the external name space. The configuration librarian's file has the same symbols, but using the "REF" construct instead. "REF" (reference) requests must be matched to corresponding definitions by the linkage editor, or an error will be reported. This provides us with a plug-socket positive version identification system. If the wrong version of a module is linked in, the linker will be unable to satisfy a "REF" request and will notify the user. This notification is not fatal; in fact, individual developers should expect to see messages for the particular modules for which they are creating new versions.

SEPARATE COMPILATION



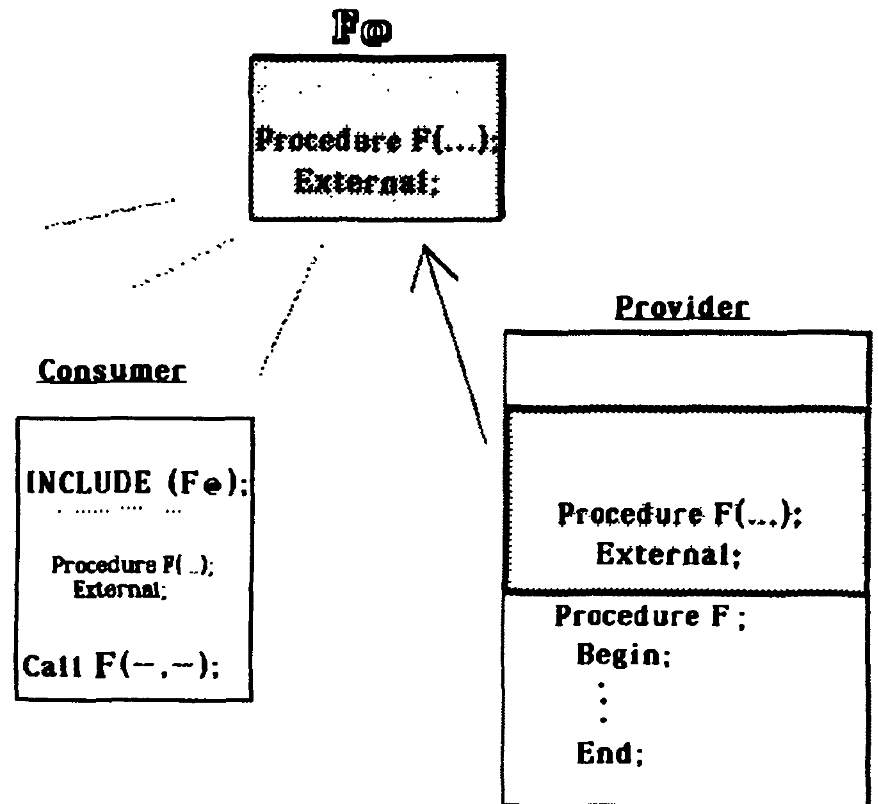
INTERFACE CONSISTENCY?

Provide "EXTERNAL" declaration for caller.

WHENCE?

- hand written
- global decl. file or from Provider

Figure 7.



INTERFACE DEFINITION OWNED BY MODULE IT DESCRIBES --

DECENTRALIZED

MANUALLY MAINTAINED BUT PHYSICALLY CLOSE

Figure 8.

Separate Compilation

There are two related problem areas involved in successfully supporting separate compilation: external interface specification, and recompilation.

In a system with hundreds of independently compiled modules, the complexity of the external interface structure becomes very high. Its correctness is vital.

The external interface allows the compiler to correctly process references which cross compilation unit boundaries. In order to process any procedure call, the compiler usually needs to see a declaration of the procedure, which will give information like the number and type of the arguments are. Often the procedure definition is "external"; i.e., it resides in another, separately compiled module. In some languages, the compiler ducks its responsibilities, and simply assumes that calls of external procedures are correct; this level of checking is not adequate for a modern, strongly-typed language. In languages such as Ada, these external declarations are provided automatically through a database maintained by the compiler. With compilers like Pascal/VS, all procedure calls are checked for validity, but the programmer has to himself provide explicit "EXTERNAL" declarations. (See figure 7.)

The challenge,then, is to create a semi-automated system to maintain the external declaration information.

In our environment, the interface was taken from a standard file, part of the general configuration managed library. The name of the interface file is derived from the file name of the provider file (by adding an "@" suffix), and so is easily and uniformly accessible. (See figure 8.) The interface information itself was extracted by a simple tool from the actual file of the provider, thus maintaining the greatest possible degree of accuracy.

It turns out that in our implementation language, it is valid to include an "EXTERNAL" declaration in the same compilation as the actual procedure definition; EXTERNAL behaves much like FORWARD. If that's not valid, the extraction system must be a bit more clever. The point is that in any case the external interface definition sits physically right next to the procedure it describes. Even if it's manually maintained, the chances of it being correct (meaning, especially, up to date) are a whole lot better than if its off on another file someplace, combined with other such interfaces. Also a feature here that we have one interface file per source module. That makes for a lot of interface files, but gives much better management of recompilation.

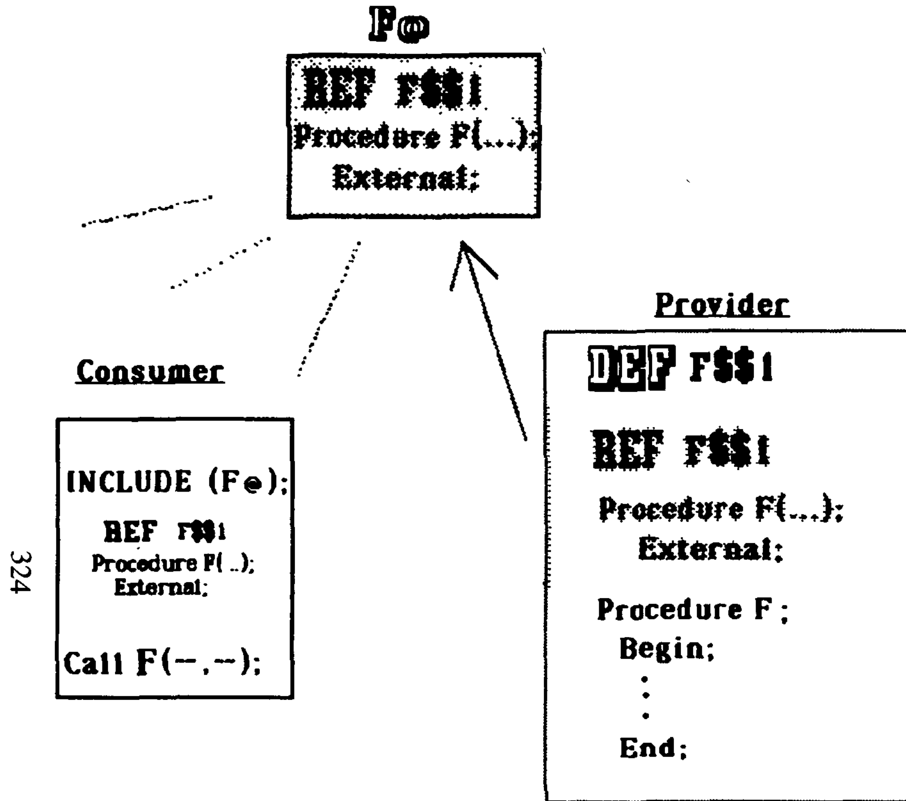


Figure 10.

RECOMPILATION

NECESSARY and SUFFICIENT CONDITIONS

NOT: just to be sure

INTERFACE CHANGE ⟹ RECOMPILE ALL USERS

NOT MODULE CHANGE

BUT:

only if change can affect them

Some additions are known to be safe

FIRM TRACKING --

use $$ symbols

+

linker diagnositics

Figure 9.

Recompilation

Maintaining correct external interface files is only half of the battle; the other is insuring that any modules which use these files are recompiled when a change occurs. For a collection of reasons, one cannot simply recompile all consumer modules every time an interface changes. The environment must, then, provide ways of tracking what has been done, and in particular of determining whether all necessary recompilations have taken place. A variation on the self-identification system provided this support.

Interface mismatches, resulting from improper recompilation, cause particularly obscure bugs. In the absence of positive accountability, one typically goes back to zero and recompiles everything when such a problem is suspected.

As we did for version control, we added a special symbol, whose name captures the module name and an interface version number. (These symbols are written with a double $$ to distinguish them from the basic version control symbols). The provider file carries the "DEF" for the symbol. It also carries a "REF", which is situated in the part of the source marked as being external interface. The extracted file, therefore, has only one of the pair, the "REF", and this shows up in every consumer. (See figure 9.)

These symbols are manually maintained. When the developer makes a change to the interface which will require recompilation of the consumers, s/he is required to increment the interface version ($$) symbol. (Using simple source file comparison tools, the configuration librarian can check whether the "$$" symbol was appropriately changed, providing an extra level of checking.) The change is made in the provider file, of course, and so only the new symbol will be defined at link time. If there are any object modules present that were compiled with the old interface, they will still carry the request for the old interface symbol. This is visible to various tools, and in particular will cause a linker diagnostic.

The use of external names for version tracking, though a simple scheme, is very powerful. It provides, inexpensively, very important information and control which are missing in many development environments.

Minimizing Recompiliation

The necessary condition for recompilation is an interface change. However, not all interface changes require recompilation; some changes can be known to be safe. In particular, in our development style the interface to a

particular module tends to consist of a large number of procedures. Adding another function is safe, as binding is done at link time. The old consumers, who don't know about the new function, aren't affected. (A careful choice of name conventions avoids the possibility of introducing name conflicts here.) As a result, we achieved a considerable resource saving, especially when dealing with key modules which are included by almost everyone.

We handled these benign interface changes simply by not changing the interface version symbol. We did not forgo version skew protection, though. If the [new] consumer module were inadvertently linked with an old version of the provider, no definition would be found for the missing functions, and a linker diagnostic would be issued.

Conclusions

This development environment proved a qualified success. It provided, as planned, very good accountability, good control of external interfaces and recompilation. It managed complexity well. But there were some loose ends, especially involving secondary interactions in the area of separate compilation. In addition, there was a lot more complexity present than had been anticipated.

In a peaceful development process, an interface revision originates in the provider module. As the changes are completed, a new interface file is created, the consumer modules modified as necessary, and then all are accepted into a new overall configuration. Under the pressure of time and multiple parallel paths of development, conflicting requirements made it difficult to adhere to a simple, orderly procedure.

- A consumer module may be temporarily unavailable, as some other developer has it signed out for other work. If simple recompilation is all that is required, that can be done from the library copy. If editing changes are needed, a severe contention problem exists. It may even be necessary to insert a new temporary version ahead of the main new version of the consumer in order to achieve a compatable whole.

- The revision may be driven by the consumer, which needs some new functionality; this may be from a module maintained by someone else. This leads to a situation in which one person is relying on temporary, private interface files taken from another person's development copy. The consumer module cannot, of course, be presented to the configuration librarian until the provider module is also ready. Our system could have provided better tracking of the version dependencies here.

The transitive nature of dependencies would sometimes lead to severe contention problems here. If A depends on B, and B on C, a change to C does not in general affect A. However, A might request a [trivial] new function from B, which B's maintainer cheerfully provides. However, B itself is in the process of incorporating some new functionality from C. Even though the A - B interface is operational, the new version of A has to be held up until the new C is available since otherwise B will be incompatible. Sometimes it becomes necessary to accept the new C, even though it does not work properly, in order that work may proceed on A and B.

These problems are beyond the original design considerations. To some extent, they are inevitable with a large project. There were also aggravated by the existence of a large, distributed interface structure in the program being developed. The problems relate to resource utilization, which are essentially management issues. Their solution lies, then, lies in providing better support for the scheduling of development so as to avoid the worst contention situations. The support environment should contribute information such as a graphical map of inter-module dependencies.

ATTENDANCE LIST 1984

BILL AGRESTI
CSC

ED ALBRIGO
SPERRY CORP

D.J.ALLEY
LOCKHEED

TROY AMES
NASA/GSFC

JACQUELINE AMRHEIN
NSA

BOB ARNOLD
MITRE CORPORATION

PATRICIA ASTILL
NASA/GSFC

EVERETT AYERS
ARINC RESEARCH CORP

CURTISS BARRETT
NASA/GSFC

DON BARRON
NSA

JEROME BARSKY
BENDIX

C.WRANDLE BARTH
NASA/GSFC

LYNN BARTON
LOCKHEED

VIC BASILI
UNIV OF MD

JOHN BAUMERT
SPACE TELESCOPE SCI INST

PETER BELFORD
CSC

BILL BELLAND
FCC

JIM BENNETT

ROY BOND
NSA

HELEN BONK
NASA/GSFC

DAVID BOON
CSC

JOHN BOWEN
HUGHES-FULLERTON

ROYCE BRADSHAW
SOCIAL SECURITY ADMIN

DAVE BRADY
TRW

MIMI BREDESON
SPACE TELESCOPE SCI INST

DALE BRENNEMAN
TRS

ELIZABETH BRINKER
NASA/GSFC

NANDER BROWN
FREDDIE MAC

DAVID BRYCH
ANALYTIC SCIENCE CORP

JOHN BUELL
CSC

ELIZABETH BUIE
CSC

CAROL BURNS
IIT RESEARCH INST

ATTENDANCE LIST 1984

DAVID CALLENDER
JET PROPULSION LAB

J. CAMPBELL
EPA

DU CAN CHAN

J.CARLSON
OAO CORP

DAVID CARD
CSC

JOHN CARL
NASA/GSFC

LLOYD CARPENTER
NASA/GSFC

MARGARET CHASSON
IBM

STEVE CHEUVRONT
CSC

ANDREW CHUNG
FAA TECH CENTER

LEE CISNEY
NASA/GSFC

JUDITH CLAPP
MITRE CORPORATION

MARVIN CLEMMONS
NASA/LANGLEY

TED CONNELL
NASA/GSFC

HARRY COOK
FED HOME MORTGAGE LOAN CO

LAURA COOK
GSC

PERRY COPP
FAA TECH CENTER

CARL CORNWELL
BENDIX

CLYDE CRAIG
AUTOMETRIC INC

STEWART CRAWFORD
BELL LABS

WILLIAM DECKER
CSC

DICK DEMEESTER

CHARLES DICKSON
USDA-ARS-CDSD

D.DILTS
AMERICAN SYS CORP

KEITH DIMORIER
NASA/JSC

DAVID DISKIN
U S CENSUS BUREAU

BERNARD DIXON
NASA/GSFC

MARYANN DOIRON
IIT RESEARCH INST

FRANK DOUGLAS
PROF S/W SERVICES

JOHN DUKE
DOD PESO

LORRAINE DUVAL
IIT RESEARCH INST

MARGARET EATON
CSC

ATTENDANCE LIST 1984

H.O.EBERHART

BETSY EDWARDS
NASA/GSFC

JENNIFER ELGOT
UNIV OF MD

DEAN ELLIOTT
NASA/GSFC

WALTER ELLIS
IBM FSD HEADQUARTERS

HARRY EMERSON
ANALYTIC SCIENCE CORP

EUNICE ENG
NASA/GSFC

MARY ANN ESTANDIAU
NASA/GSFC

MIKE FAGAN
UNIV OF MARYLAND

HOSEIN FALLAH
AT&T BELL LABS

AI FANG
NASA HEADQUARTERS

WILLIAM FARR
NAVAL SURFACE WEAPONS CTR

JAMES FARRELL
WESTINGHOUSE

RICHARD FATH
FCC

LARRY FISHTAHLER
CSC

CELIA FITZERALD
DATA GENERAL CORP

WAYNE FRIEDMAN
B.F.E.C.

JOHN GAFFNEY
IBM CORP

JULIA MEADE GALLIER
NAVAL SURFACE WEAPONS CTR

PETER GAMM
PRC

JOSEPH GARNER
ADV COMP SCI GROUP

PAT GARY
NASA/GSFC

C.H.GAUDETTE
IBM

RICHARD GAYLE
JOINT TACTICAL COMMAND CONTROL
& COMMUNICATION AGENCY

DIETWALD GERSTNER
NASA HEADQUARTERS

NORMAN GLICK
NSA

JOHN GOLDEN
EASTMAN KODAK

ADOLF GOODSON
NASA/GSFC

CAROLINE GRAFTON
DEFENSE SYSTEMS INC

ROBERT GRAFTON
OFFICE OF NAVAL RESEARCH

ART GREEN
CSC

ATTENDANCE LIST 1984

SCOTT GREEN
NASA/GSFC

ARNOLD GREENLAND
IIT RESEARCH INST

STEPHEN GREIF
BENDIX

DR.C.J.GREWE
MARTIN MARIETTA AEROSPACE

DAVID HAMMEN

DICK HANKINS

MYRON HECHT
SOHAR INCORP

J.L.HECK
SOHAR INCORP

CARL HEISE
FCC

DOUGLAS HILLMER
CENSUS BUREAU

BARBARA HOLMES
GSC

ROBERT HOLT
GEORGE MASON UNIV

ROD HOUSTON
IIT RESEARCH INST

ALAN HOWLETT
IIT RESEARCH INST

LARRY HULL
NASA/GSFC

NORMAN IDELSON
IIT RESEARCH INST

MARY ELLEN INGHAM
NSA

DONALD JENKINS
FAA

DAVID JOESTING
BFEC NAO/SM

LEON JORDAN
CSC

LINDA JUN
NASA/GSFC

DENNIS KAFURA
VA POLYTECHNIC INST

OWEN KARDATZKE
NASA/GSFC

ELIZABETH KATZ
UNIV OF MD

FRANCES KAZLAUSKI
NSA

JOE KELLAGHER
US DEPT OF COMMERCE

JOHN KNIGHT
UNIV OF VIRGINIA

KATHY KOERNER
CSC

RICHARD KOPKA
DOD/ECAC

PATTY KRAMER
EPA

THOMAS KURIHARA
US DEPT OF TRANS

DAVID LAME
JET PROPULSION LAB

ATTENDANCE LIST 1984

DICK LANGLEY
FCC

NANCY LAUBENTHAL
NASA/GSFC

MARGARET LAVIGNE

RAY LEBER
GENERAL ELECTRIC

GERTRUDE LEE
DOTY ASSOCIATES

DAVID LEVINE
INTERMETRICS

ROSCOW LIN
LOCKHEED

JANET LINDGRAN
INFOMATICS

JEAN LIU
CSC

KUEN-SAN LIU
CSC

PEI-SHEN LO
CSC

MICHELLE LOOSER
FAA TECH CENTER

JANET LUNDGREN
INFORMATICS GENERAL CORP

BILL MADDOX
GENERAL DYNAMICS

JOHN MANLEY
NASTEC CORP

THOMAS MASTERS
NSA

RAY MAZZOLA
FORD AEROSPACE

TOM MCCABE
MCCABE ASSOC

S.MCCARRON
NASA/GSFC

W.L.MCCOY
NSWC

FRANK MCGARRY
NASA/GSFC

MARY ANN MCGARRY
IITRI

DANIEL MCGOVERN
FAA TECH CENTER

JOHN MCLEOD
JET PROPULSION LAB

EDWARD MEDEIROS
CSC

REG MEESON
COMPUTER TECHNICAL ASSOC

MICHAEL MELCHIORRE
BURROUGHS CORP

ROBERT MEMBRINO
SINGER CO

VICTORIA MENDENHALL
NAVAL SURFACE WEAPONS CTR

PHIL MERWARTH
NASA/GSFC

MARY LOU MIDDLETON
FCC

ATTENDANCE LIST 1984

TIM MILES
U.S.DEPT OF COMMERCE

WARREN MILLER
CSC

TAWNA MINTON
NSA

RAKESH MITAL
CSC

KAREN MOE
NASA/GSFC

S. MOHANTY
MITRE CORP

EILEEN MUNDAY
CSC

ROBERT MURPHY
NASA/GSFC

MARY MYERS
BURROUGHS CORP

PHILIP MYERS
CSC

AHMED NADEEM
MITRE CORP

MATT NADELMAN
CSC

DAVID NADOLNA
NSA

DEBRA NADOLNA
NSA

JOE NAPKURI

BURT NEWLIN
DMSSO

ED NG
JET PROPULSION LAB

ROBERT NOONAN
WILLIAM & MARY

DR.A.F.NORCID
NAVAL RESEARCH LAB

JANE OHLMACHER
SOCIAL SECURITY ADMIN

L. O'NEIL
BELL LABS

JERRY PAGE
CSC

ROGER PANARA
RADC/COEE

NIKKI PANLILTO-YAP
UNIV OF MD

EDDIE PARAMORE

PAUL PASHBY
NASA/GSFC

TERESA PASSALACQUA
CENSUS BUREAU

MICHAEL PASTERNAK
NAVAL SURFACE WEAPONS CTR

DEBA PATNAIK
UNIV OF MD

MICHAEL PATTON

RAYMOND PAUL
NAVAL SEA SYSTEMS COMMAND

LEONIE PENNEY
PENNEY ASSOCIATES

ATTENDANCE LIST 1984

DOLLY PERKINS
NASA/GSFC

GIOVANNI PERRONE
MARTIN MARIETTA AEROSPACE

KARL PETERS
NASA/GSFC

B.JANE PETERSON
AUTOMETRIC INC

CAROL PETROSKI

JOHN PIETRAS
MITRE CORP

MICHAEL PLETT
CSC

WILLIAM POSTHUMA
NASA/GSFC

WILLIAM POW

DAVID PRESTON
IITRI

DOUGLASS PUGH
IITRI

DOUGLAS PUTMAN
QSM

LARRY PUTMAN
QSM

JOHN QUANN
NASA/GSFC

THOMAS QUINDRY
DOD PESO

CONNIE RAMSEY
UNIV OF MARYLAND

JAMES RAMSEY
UNIV OF MD

DR.ED RANG
HONEYWELL

GEORGE RATTE
USDA-ARS

JOHN REDDING
FEDSIM/CAA

DONALD REIFER
REIFER CONSULTANTS INC

PAT RINN
FCC

DON ROBBINS
NSA

RICHARD ROBINSON
MITRE CORP

H. DEITER ROMBACH
UNIV OF MD

JORGE ROMEU
IITRI

ROBERT ROSSIN
GENERAL ELECTRIC CO

DAN ROY
CENTURY COMPUTING

JOYCE RUETHGER
MCCOHEN & ASSOC

STEVE RUGALA
FCC

ROMAINE RUPP
BURROUGHS CORP

VINCIENT RUPOLO
BANKERS TRUST CO

ATTENDANCE LIST 1984

ANDY RUTHERFORD

JOAN SANBORN
NASA/GSFC

CATHRYN SAVOLAIN
BELL LABS

BOB SCHWENK
NASA/GSFC

RICHARD SELBY
UNIV OF MD

ED SEIDEWITZ
NASA/GSFC

EDMOND SENN
NASA/LANGLEY

PAUL SERAFIN
EG&G

SILVIA SHEPPARD
COMPUTER TECH ASSOC

ARNOLD SMITH
MARTIN MARIETTA AEROSPACE

JOHN SMITH
NAVAL SURFACE WEAPONS CTR

OLIVER SMITH
EG & G

PATRICIA SMITH
NSWC

GLENN SNYDER
CSC

MARIA SO
NASA/GSFC

DAVID SOLOMAN
CSC

DUANE SOSKEY
CSC

C.B.SPENCE
CSC

AL STAMENT
PRC SYSTEMS SERVICE

MIKE STARK
NASA/GSF

JODY STEINBACHER
JET PROPULSION LAB

BARBARA STONE
PRC SYSTEMS SERVICE

RAY SUCHY
NSA

STEVE SUDDITH
GSC

JUDIN SUKRI
UNIV OF MD

ROBERT SUM
UNIV OF ILLINOIS

STEVE SWARTZ
FCC

M.LISA SYLLA

DEBRA SYNOTT
JET PROPULSION LAB

PAUL SZULEWSKI
DRAPER LABS

N.TAMARCHENKO
DATA GENERAL CORP

KEIJI TASAKI
NASA/GSFC

ATTENDANCE LIST 1984

KENNETH TOM
ARINC

CAROL URT
FCC

JOHN UTZ
EG & G

ALAN VAN BOVEN
GTE SYSTEMS

JON VALETT
NASA/GSFC

NANCY VEILLON
CSC

SUSAN VOIGT
NASA/LANGLEY

DOLORES WALLACE
NAT BUREAU OF STANDARDS

DAVID WEISS
US NAVAL RESEARCH LAB

PETER WEISS
COMSAT

LT.GREG WELZ
AFSTC/VLC

MARILEE WHEATON
AEROSPACE CORP

KEN WILLIAMS
PLANNING RESEARCH CORP

STEVE WILLIAMS
TRW

RAY WOLVERTON
ITT PROGRAMMING TECH CTR

ALICE WONG
FED AVIATION ADMIN

CHARLES YOUMAN
CEY ENTERPRISES

ANDREW YOUNG
LOCKHEED

LEON YOUNG
REIFER CONSULTANTS INC

LARRY ZEIGENFUSS
NASA/GSFC

MARV ZELKOWITZ
UNIV OF MARYLAND

PRANAS ZUNDE
GEORGIA TECH UNIV

ART ZYGIELBAUM
JET PROPULSION LAB

STANDARD 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

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-78-102, FORTRAN Static Source Code Analyzer Program (SAP) User's Guide (Revision 1), W. J. Decker and W. A. Taylor, September 1982

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

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

SEL-80-104, Configuration Analysis Tool (CAT) System Description and User's Guide (Revision 1), W. Decker and W. Taylor, December 1982

SEL-81-006, Software Engineering Laboratory (SEL) Document Library (DOCLIB) System Description and User's Guide, W. Taylor and W. J. Decker, December 1981

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

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-81-101, Guide to Data Collection, V. E. Church, D. N. Card, F. E. McGarry, et al., August 1982

SEL-81-102, Software Engineering Laboratory (SEL) Data Base Organization and User's Guide Revision 1, P. Lo and D. Wyckoff, July 1983

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

SEL-81-107, Software Engineering Laboratory (SEL) Compendium of Tools, W. J. Decker, W. A. Taylor, and E. J. Smith, February 1982

SEL-81-110, Evaluation of an Independent Verification and Validation (IV&V) Methodology for Flight Dynamics, G. Page and F. McGarry, December 1983

SEL-81-203, Software Engineering Laboratory (SEL) Data Base Maintenance System (DBAM) User's Guide and System Description, P. Lo, June 1984

SEL-81-205, Recommended Approach to Software Development, F. E. McGarry, G. Page, S. Eslinger, et al., April 1983

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

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-82-105, Glossary of Software Engineering Laboratory Terms, T. A. Babst, F. E. McGarry, and M. G. Rohleder, October 1983

SEL-82-206, Annotated Bibliography of Software Engineering Laboratory Literature, D. N. Card, Q. L. Jordan, and F. E. McGarry, November 1984

SEL-83-001, An Approach to Software Cost Estimation, F. E. McGarry, G. Page, D. N. Card, et al., February 1984

SEL-83-002, Measures and Metrics for Software Development, D. N. Card, F. E. McGarry, G. Page, et al., March 1984

SEL-83-003, Collected Software Engineering Papers: Volume II, 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-83-104, Software Engineering Laboratory (SEL) Data Base Retrieval System (DARES) User's Guide, T. A. Babst, W. J. Decker, P. Lo, and W. Miller, August 1984

SEL-83-105, Software Engineering Laboratory (SEL) Data Base Retrieval System (DARES) System Description, P. Lo, W. J. Decker, and W. Miller, August 1984

SEL-84-001, Manager's Handbook for Software Development, W. W. Agresti, V. E. Church, and F. E. McGarry, April 1984

SEL-84-002, Configuration Management and Control: Policies and Procedures, Q. L. Jordan and E. Edwards, December 1984

SEL-84-003, Investigation of Specification Measures for the Software Engineering Laboratory (SEL), W. Agresti, V. Church, and F. E. McGarry, December 1984

SEL-RELATED LITERATURE

Agresti, W. W., Definition of Specification Measures for the Software Engineering Laboratory, Computer Sciences Corporation, CSC/TM-84/6085, June 1984

[1]Agresti, W. W., F. E. McGarry, D. N. Card, et al., "Measuring Software Technology," Program Transformation and Programming Environments. New York: Springer-Verlag, 1984

[2]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

[2]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)

[2]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

[2]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

[1]Basili, V. R., and B. T. Perricone, "Software Errors and Complexity: An Empirical Investigation," Communications of the ACM, January 1984, vol. 27, no. 1

[2]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

[1]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 J. Ramsey, Structural Coverage of Functional Testing, University of Maryland, Technical Report TR-1442, September 1984

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

[1]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

[2]Basili, V. R., and M. V. Zelkowitz, "Operation of the Software Engineering Laboratory," Proceedings of the Second Software Life Cycle Management Workshop, August 1978

[2]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

[2]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]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

[1]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

Higher Order Software, Inc., TR-9, A Demonstration of AXES for NAVPAK, M. Hamilton and S. Zeldin, September 1977 (also designated SEL-77-005)

Page, G., F. E. McGarry, and D. N. Card, "A Practical Experience With Independent Verification and Validation," Proceedings of the Eighth International Computer Software and Applications Conference, November 1984

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

[2]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

[1]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 article also appears in SEL-83-003, Collected Software Engineering Papers: Volume II, November 1983.

[2]This article also appears in SEL-82-004, Collected Software Engineering Papers: Volume I, July 1982.