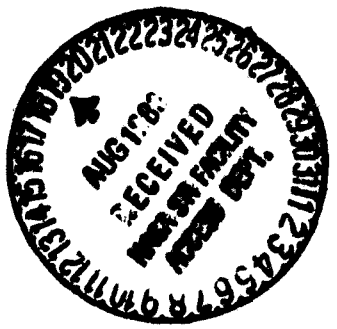


PROCEEDINGS OF THE SEVENTH ANNUAL SOFTWARE ENGINEERING WORKSHOP

DECEMBER 1982

National Aeronautics and
Space Administration

Goddard Space Flight Center
Greenbelt, Maryland 20771

PROCEEDINGS

OF

SEVENTH ANNUAL SOFTWARE ENGINEERING WORKSHOP

Organized by:

Software Engineering Laboratory
GSFC

December 1, 1982

GODDARD SPACE FLIGHT CENTER
Greenbelt, Maryland

FOREWORD

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

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

The goals of the SEL are (1) to understand the software development process in the GSFC environment; (2) to measure the effect of various methodologies, tools, and models on this process; and (3) to identify and then to apply successful development practices. The activities, findings, and recommendations of the SEL are recorded in the Software Engineering Laboratory Series, a continuing series of reports that includes this document. A version of this document was also issued as NASA/GSFC document in 1982.

Single copies of this document can be obtained by writing to

Frank E. McGarry
Code 582.1
NASA/GSFC
Greenbelt, Maryland 20771

SEVENTH ANNUAL SOFTWARE ENGINEERING WORKSHOP

ABOUT THE WORKSHOP

The Seventh Annual Software Engineering Workshop was held on December 1, 1982, at Goddard Space Flight Center in Greenbelt, MD. Nearly 250 people, representing 9 universities, 22 agencies of the federal government, and 43 private organizations, attended the meeting.

As in the past 6 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 7 years (1976 to 1982), 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 582.1
NASA/GSFC
Greenbelt, MD 20771

301-344-5048

AGENDA

SEVENTH ANNUAL SOFTWARE ENGINEERING WORKSHOP
NASA/GODDARD SPACE FLIGHT CENTER
BUILDING 3 AUDITORIUM
DECEMBER 1, 1982

8:00 a.m.	Registration "Sign-In" Coffee-Donuts	
8:30 a.m.	INTRODUCTORY REMARKS "What Have We Learned in 6 Years?"	F. E. McGarry (NASA/GSFC)
9:00 a.m.	SESSION NO. 1	TOPIC: The Software Engineering Laboratory (SEL)
		Discussant: J. Page (CSC)
	"Software Errors and Complexity, An Empirical Investigation"	V. Basili (University of MD)
	"When and How to Use a Software Reliability Model"	A. Goel (Syracuse University)
	"Measuring the Application of Software Prototypes"	M. Zelkowitz (University of MD)
10:30 a.m.	BREAK	
11:00 a.m.	SESSION NO. 2	TOPIC: Software Tools
		Discussant: P. Scheffer (Martin Marietta)
	"Experience and Perspectives with SRI's Tools for Software Design and Validation"	J. Goguen (SRI) K. Levitt (SRI)
	"Technology Transfer Software Engineering Tools"	I. Miyamoto (University of MD)
	"Design Aids for Real-Time Systems"	P. Szulewski (Draper Labs)
12:30 p.m.	LUNCH	

1:30 p.m.	SESSION NO. 3	TOPIC: Software Errors
		Discussant: D. Simkins (IBM)
	"Software Error Data Collection and Categorization"	T. Ostrand (Sperry Univac) E. Weyuker (Courant Inst.)
	"An Effective Bug Classification Scheme Must Take the Programmer Into Account"	E. Solloway (Yale) W. Johnson (Yale) S. Draper (University of CA)
	"Software Anomoly Taxonomy What Can be Gained?"	D. Buckland (Reifer Consultants)
3:00 p.m.	BREAK	
3:30 p.m.	SESSION NO. 4	TOPIC: Cost Estimation
		Discussant: D. Card (CSC)
	"Maintenance Estimation Methodology"	K. Rone (IBM)
	"Staffing Implications of Software Productivity Models"	R. Tausworthe (JPL)
	"Estimates of Software Size From State Machine Designs"	R. Brithcher (IBM) J. Gaffney (IBM)
5:00 p.m.	ADJOURN	

SUMMARY OF THE SESSIONS: SEVENTH ANNUAL SOFTWARE ENGINEERING WORKSHOP

Michael Rohleder

COMPUTER SCIENCES CORPORATION

and

THE GODDARD SPACE FLIGHT CENTER

SOFTWARE ENGINEERING LABORATORY

Prepared for the

NASA/GSFC

Seventh Annual Software Engineering Workshop

December 1982

INTRODUCTORY REMARKS

Frank McGarry - "What Have We Learned in Six Years?"

Frank McGarry of the Goddard Space Flight Center (GSFC) opened the workshop with a summary of results obtained from the analysis of data collected by the Software Engineering Laboratory (SEL). The SEL has monitored 46 software development projects at GSFC during the past 6 years. The discussion covered the areas of profiles, models, and methodologies. Within these areas, a number of results were presented.

The use of modern programming practices (MPP) favorably affects productivity and reliability. A 15-percent increase in productivity was demonstrated. However, the effect of MPP on reliability was found to be highly variable. Programmer ability and experience was shown to have the greatest influence on the productivity of the software development process. Studies of reliability and cost models were inconclusive. More theoretical development of and practical experience with such models is needed before they can be applied effectively in a production environment.

The costs of data collection were identified and quantified. These include task overhead, data processing, and data analysis. Data collection is expensive, but it is essential to understanding and improving the software development process.

In response to questions and comments from the audience, McGarry clarified several points:

- A number of methodologies have proved to be cost effective in the GSFC environment. However, numerical values for the benefits and costs of

individual methodologies are difficult to determine. The maximum savings observed were about 15 to 20 percent for a combination of MPP.

- Except for errors, data from the maintenance phase was not included in these analyses.

SESSION 1 - THE SOFTWARE ENGINEERING LABORATORY

Victor Basili--"Software Errors and Complexity, An Empirical Investigation"

The first speaker of the first session was Victor Basili of the University of Maryland. This presentation focused on the distributions and relationships derived from error data collected during the development of a medium-scale software project. The error characteristics of this project were shown to reflect significant differences between this project and the class of projects usually studied by the SEL.

Modified and new modules were shown to differ in the types of errors prevalent in each and the amount of effort required to correct an error. Modified modules appeared to be more susceptible to errors due to the misunderstanding of specifications. One surprising result presented by Basili was that an increase in module size did not increase error proneness. In fact, larger modules were shown to be less error prone. This was true even though the larger modules were more complex. A number of explanations for this phenomenon were suggested.

In response to questions and comments from the audience, Basili clarified the following points:

- Errors of commission were those errors caused by an incorrect program statement. Errors of omission were those errors that resulted from forgetting to include a statement or parameter.
- A large portion of the errors was attributed to a misunderstanding of specifications or requirements.
- The effect of programmer experience was considered in the investigation.

- Additional work is required to determine the optimum size of modules with respect to reliability.
- Errors caused by earlier error correction efforts were found to be, at most, 6 percent of the total.
- Data was not available on the time required to correct errors in large versus small modules.

Amrit Goel--"When and How To Use a Software Reliability Model"

The second speaker of the session was Amrit Goel from Syracuse University (on leave to the University of Maryland). This presentation dealt with the role of software errors in determining the reliability of large-scale, computer-based systems. The use of stochastic and combinatorial models to assess system reliability in the presence of failures caused by software errors was examined. It was suggested that users were employing models that were readily available on their computer systems rather than the most appropriate model for their development environments. This is due to incorrect or ambiguous interpretations of model assumptions and output.

Goel presented views about the utility of the available models during various stages of the development process and in different testing situations. Alternatives to reliability models were also suggested for occasions when the currently available models do not seem to be applicable.

The following points were made by the audience in response to the presentation:

- Rick Gale pointed out that software testing should be driven by reliability model measures.
- John Musa agreed that appropriate testing is necessary to obtain valid results from a model.

Marvin Zelkowitz--"Measuring the Application of Software Prototypes"

The last speaker of the first session was Marvin Zelkowitz of the University of Maryland. This presentation covered the development and application of prototypes for software systems. The differences between models and prototypes were identified as well as essential elements common to both. Environmental considerations and their influences on prototype development were also discussed.

An ongoing experiment in prototyping, the Flight Dynamics Attitude Simulator (FDAS), was described. A number of factors motivated the choice of the prototyping approach for the development of this system. These include uncertainties about size, requirements, and interfaces.

In response to questions and comments from the audience, Zelkowitz clarified the following points:

- The major goal in the development of this prototype is to examine project requirements and feasibility more closely. Specifications for the full system will be based on the results of the prototyping experience.
- The need for prototype development stems from the fact that FDAS is a very different type of system from those usually developed in this environment.
- Prototypes are not built merely to "tack on" additional features at a later date to build the full system. Some elements may migrate to the full system, however.
- Elaine Weyuker disagreed with the 10-percent estimate for the cost of a prototype versus full implementation and suggested that 30 percent is more realistic in a nonacademic environment.

SESSION 2 - SOFTWARE TOOLS

Karl Levitt and Joseph Goguen--"Experience and Perspectives With SRI s Tools for Software Design and Validation"

The initial speakers of the second session were Karl Levitt and Joseph Goguen from SRI International. The joint presentation described current approaches to software tools for design specification and presented experiences with several projects at SRI.

Four development tools were introduced: the STP theorem prover and its associated Design Verification System; PHIL, a meta-programmable, context-sensitive structured editor; Pegasus, a system for supporting graphics programming; and OBJ, an ultra-high-level programming language based on rewrite rules and abstract data types.

The speakers described successful efforts to apply these tools to design specification and verification for two classes of systems in which reliability is vital: fault-tolerant systems for aircraft control and secure operating systems.

In response to questions and comments from the audience, the following points were clarified:

- A major purpose of a specification language is to support the decomposition and testing of designs at an early stage.
- The most compelling reason for the lack of formal specifications languages with tool support is the absence of examples that model good specifications having the right amount of detail.

Isao Miyamoto--"Technology Transfer Software Engineering Tools"

The second speaker of the session was Isao Miyamoto from the University of Maryland, Baltimore County, who discussed technology transfer as it applies to software engineering tools.

Experiences with tool usage and availability were presented. Miyamoto identified three reasons that tools are not used:

1. Lack of a clearly defined methodology
2. Economic ineffectiveness
3. Lack of measures and criteria for evaluating the effectiveness of tools

An example was presented of a software maintenance support tool system called "Pandora's Box." This system provides users with a hierarchical network of menus designed to provide user-friendly capabilities from novice to expert. It is hoped that the project will produce a tool that will gain user acceptance.

In response to a question from the audience, Miyamoto clarified the following point: designing easy-to-use, cost-effective tools is the key point in transferring software engineering technology from the research laboratory to users.

Paul Szulewski--"Design Aids for Real-Time Systems"

The last speaker of the session was Paul Szulewski of the Draper Laboratory. The presentation described ongoing efforts with Design Aids for Real-Time Systems (DARTS). This tool assists in defining embedded computer systems through tree-structured graphics, military standards documentation support, and various analyses including calculation of Halstead's Software Science measures.

DARTS uses a mix of hierarchical organization, control conventions, communications primitives, and data structures to represent real-time systems. Requirements are expressed as a functional hierarchy, and the design is represented as a tree-structured hierarchy of communicating processes.

Through a user-friendly, menu-oriented interface, a user can define a system; perform data flow checking; generate simulations of response time, throughput, and utilization; request a variety of data tables and graphical tree-structured output in various sizes; and calculate Software Science measures.

In response to questions and comments from the audience, Szulewski clarified the following points:

- DARTS is operational on an Amdahl 470 V8. It consists of approximately 20,000 lines of PL1 code.
- DARTS has not been used thus far for applications such as PERT charting.
- Tool availability and desirability from a user's standpoint are important aspects of tool design.

SESSION 3 - SOFTWARE ERRORS

Thomas Ostrand - "Software Error Data Collection and Categorization"

The first speaker of the third session was Thomas Ostrand of Sperry Univac, who presented the results of a research project done jointly with Elaine Weyuker. The project analyzed the relationship of error characteristics to various aspects of the software development process by studying software errors committed during the development of an interactive, special-purpose editor system. A new error categorization system was developed and 174 errors were classified with this scheme.

The new error categorization scheme was developed from programmer descriptions of errors, their symptoms, and corrections. Four generic attributes, or dimensions, of software errors were identified; each error was classified by assigning it a value for each dimension. These dimensions and their possible values reflect the specific errors identified during the project. These dimensions include major category, type, presence, and use.

In response to questions and comments from the audience, Ostrand clarified the following points:

- Good rapport with the programmers is vital to success in data collection efforts.
- Design was done informally. Flowcharts, formal requirements, and specifications were not used.
- The importance of relevant information in data collection efforts cannot be overemphasized.

Elliot Solloway--"An Effective Bug Classification Scheme Must Take the Programmer Into Account"

The next speaker of the third session was Elliot Solloway of Yale University, who presented a paper coauthored by W. Johnson, also of Yale, and S. Draper of the University of California. This presentation defined a particular view of bug classification. Rather than looking at productivity or reliability, the goal in looking at program bugs was to provide a basis for building computer-based tutoring systems that can aid the novice in learning to program. The conclusion is that bugs are not random occurrences but, rather, systematic and provide a window into misconceptions that novices have about programming.

Developing a classification scheme for bugs based solely on the surface features of the programs themselves is insufficient to uniquely classify bugs, and it ignores the underlying misconception. What is needed are heuristic rules based on a hypothesis of what the programmer's intentions were as he she created the program. Classifying bugs must take the programmer into account.

In response to questions and comments from the audience, Solloway clarified the following points:

- Careless programming practices produce more errors in code. Classification of these errors becomes increasingly more difficult as the number of errors increases.
- Errors in programs can be classified using information about how they were fixed.
- Vic Basili distinguished between errors and faults. Finding a fault leads to a search for the error.

- Care must be taken to ensure the quality of data collected.

Donna Buckland--"Software Anomaly Taxonomy--What Can Be Gained?"

The last speaker of the third session was Donna Buckland of Reifer Consultants. This presentation discussed the results of a study to categorize software errors that had been reported during the stages of testing and operational use of the Deep Space Network DSN/Mark 3 system and to build a data base for subsequent analysis.

A three-dimensional classification scheme was devised to capture error data for statistical and trend analysis. These dimensions are time of occurrence, error criticality, and error category. The first dimension defines the particular software life cycle phase in which the error was introduced. Criticality assesses the severity of the error. Error category defines the cause of the error.

Buckland stated that the collection and classification of software error data provides management with a powerful tool for isolating problem areas. The data can be used to identify error-prone modules and serve as a basis for making repair and/or replacement decisions.

In response to questions and comments from the audience, Buckland clarified the following points:

- Quantification of error data is a very important tool.
- The length of time required to fix a problem is also very important and is sometimes overlooked.
- Vic Basili pointed out that it is often difficult to get an individual who fills out a change/error report to understand exactly what information is needed.

SESSION 4 - COST ESTIMATION

Kyle Rone--"Maintenance Estimation Methodology"

The first speaker of the fourth session was Kyle Rone of the International Business Machines Corporation (IBM). This presentation described a systematic approach to providing estimates for both staffing and skill levels during the maintenance phase of a project.

The approach presented uses a Rayleigh curve method of projection combined with a modified matrix method to forecast maintenance needs and required staffing levels. The curves generated by both methods are differenced to ascertain how much new work can be performed given the staffing level. Actual data is compared to projections to validate or modify the process.

In response to questions and comments from the audience, Rone clarified the following points:

- Estimation is not a one-time process; it must be applied over and over again.
- Maintenance activities include correction of both latent and ongoing errors.
- The amount of maintenance required can be reduced by applying more quality control during early development phases. Quality is cheaper in the long run.
- Frank McGarry stated that independent verification and validation (IV&V) is appropriate for projects with high reliability requirements. The effect of IV&V on maintenance costs has not been assessed by the SEL.

- Dave Card asked whether unmaintainable software has ever been encountered. The response from Rone was that such software has been encountered and must be disposed of.
- The type of model used in estimation is not as important as using a given model regularly with good techniques that are transportable.

Robert Tausworthe--"Stafting Implications of Software Productivity Models"

The second speaker of the fourth session was Robert Tausworthe of the Jet Propulsion Laboratory (JPL). His presentation investigated the implications of equating a project staffing model with an intercommunication overhead model in a small neighborhood of project effort. Highlights from the study include the following: there is a calculable maximum effective staff level for any project beyond which additional staff does not increase the production rate; this limits the extent to which effort and time may be traded effectively. It becomes ineffective in a practical sense to expend more than an additional 25 to 50 percent of resources in order to reduce delivery time. Additionally, it was pointed out that the project intercommunication overhead can be determined from the staffing level for a given project.

The following point was clarified by Tausworthe in response to a question from the audience: Dave Card asked whether intercommunication overhead could be reduced by dividing a project into a number of tasks that communicate only through the manager. Tausworthe replied that the increased management activity would increase overhead costs even faster.

John Gaffney--"Estimates of Software Size From State Machine Designs"

The final speaker of the fourth session was John Gaffney of the National Weather Service, on loan from IBM, who presented a paper coauthored by Robert Britcher of IBM. The presentation explained how the length or size of programs (in number of source lines of code) represented as state machines can be reliably estimated in terms of the number of internal state machine variables. Variables here are defined as the unique data required by a state machine's transition function, not the data retained in the state machine's memory. These are equivalent to Halstead's operands. The methodology presented can be employed at successive stages of the development process to provide increasingly accurate estimates.

The following points were made during the ensuing discussion:

- Kyle Rone asserted that cost estimation is not an exact science; it is a way of accumulating experience to make accurate estimates in a given environment.
- Dave Card suggested that different analysts might decompose a state machine model differently and thus get different results. Gaffney replied that the effect of such results could be important but that they could be minimized by careful and consistent application of the decomposition technique.

D/

WHAT HAVE WE LEARNED IN THE LAST 6 YEARS N83 32357

MEASURING SOFTWARE DEVELOPMENT TECHNOLOGY

BY

FRANK E. McGARRY
GODDARD SPACE FLIGHT CENTER

In late 1976, the Goddard Space Flight Center (GSFC) initiated efforts to create a software laboratory where various software development technologies and methodologies could be studied, measured and enhanced. This laboratory became known as the Software Engineering Laboratory (SEL), and since its inception has been actively conducting studies and experiments utilizing flight dynamics projects in a production environment. The SEL evolved to a full partnership in the efforts between GSFC, the University of Maryland and Computer Sciences Corporation (CSC).

The approach that the SEL has taken in carrying out the studies has been to apply varying methodologies, tools, management concepts, etc. to software projects at Goddard; then to closely monitor the entire development cycle so that the entire process and product can be compared to similar projects utilizing somewhat different approaches. This monitoring function led to a need to collect, store and interpret great amounts of data pertaining to all phases of the software process, product, environment and problem. This data collection and data processing process has been applied to over 40 software projects ranging in size from 2,000 lines of code to approximately 120,000 lines of code with the typical project running about 55,000 lines of code.

The data that has been collected (and is still being collected) and interpreted for these projects comes from 5 sources:

1. Data Collection forms utilized by programmers, managers and support personnel. Typical types of data collected include:

   - o Error and Change Information
   - o Weekly Hours and Resources
   - o Component Effort (hours expended on each component by week)
   - o Project Characteristics
   - o Computer Run Analysis
   - o Change and Growth History (week by week records of source code)

(Additional Information is contained in references 1 and 2)

2. Computer Accounting Information

3. Personnel Interviews-during and after the development process

4. Management and Technical Supervisor Assessments

5. Tools-used to extract data and measures from source code

For the more than 40 projects which have been monitored, approximately 21,000 forms have been processed and are continually used to perform studies of the software development process. To support the storage, validation and usage of this information, a data base was designed and built on a PDP-11/70 at Goddard. (Reference 3)

Approach (Chart 2)

The steps that have been taken to carry out the investigation within the SEL have been:

1. Develop a profile of the software development process as it is 'now'. First we must understand what we do well and what we do not so well so we can build a baseline of current characteristics whereby later we can honestly measure change.

2. Experiment with similar type projects. The second step has been to apply select tools, methodologies and approaches to software projects so they can be studied for effect.

3. Measure the process and product. As projects are developed which are utilizing different software development techniques, the SEL uses the extracted data to determine whether or not the applied technology has made any measurable impact on the software characteristics (This may include reliability, productivity, complexity, etc.).

Environment (Chart 3)

The projects which have been monitored and studied are primarily all flight dynamics related software systems. This software includes applications to support attitude determination, attitude control, maneuver planning, orbit adjust and general mission analysis.

The attitude systems normally have very similar characteristic and all are designed to utilize graphics as well as to run in batch mode. Depending on the problem characteristics, the typical attitude systems range in size from 30,000 to over 120,000 lines of code.* The percentage of reused code ranges from less than 10 percent to nearly 70, percent with the average software package being comprised of approximately 30 percent reused code.

The applications are primarily scientific in nature with moderate reliability requirements and normally are not required to run in real time. The development period typically runs for about 2 years (from Requirements Analysis through Acceptance Testing). The development computers are typically a group of IBM S/360's which have very limited resources and where reliability is quite low (typically less than 3 hours MTBF)

Details describing the environment can be found in Reference 1.

*Here, a line of code is any 80 byte record processable by a compiler or assembler (i.e., comments are included)

Experiments Completed (Chart 4)

As was mentioned earlier, the SEL has monitored over 40 software development projects during the 6 years of operation. During this time period, numerous methologies, models, tools and general software approaches have been applied and measured. The summary results to be presented are based on these projects. The summary will be divided into 3 topic areas:

1. Profiles of the Development Process
2. Models
3. Methodologies

Profiles of the Development Process (Charts 5 thru 12)

The first step in attempting to measure the effectiveness of any software technology is to generate a baseline or profile of how one typically performs his job. Then as modified approaches are attempted on similar projects, the effects may be apparent by comparison.

Resources Allocation (Chart 7)

One set of basic information that one may want to understand is just where do programmers spend their time. When the SEL looked at numerous projects to understand where the time was spent, it found that the SEL environment deviated somewhat from the old 40-20-40 rule. Typically projects indicated that when the total hours expended were based on phase dates of a project (i.e., a specific data defined the absolute completion of one phase of the cycle and the beginning of the next phase) the breakdown was less than 25 percent for design, close to 50 percent for code and about 30 percent for integration and test.

When the programmers provided weekly data attributing their time to the activity that they felt they were actually doing, no matter what phase of software development they were in; the profile looks quite different. The 3 phases (design, code, test) each consumed approximately the same percent effort and over 25 percent of the time was attributed to 'other' activities (such as travel, training, unknown, etc.). The SEL has continually found that this effort (other) exists, and cannot easily be reduced, and most probably should be accepted as a given. The SEL has found it to be a mistake to attempt to increase productivity merely by eliminating major portions of this 'other'time.

Development Resources (Chart 8)

Another area of concern to the SEL in defining the basic profile of software development, was that of staffing level and resource expenditure profiles. Many authorities subscribe to the point that there is an optimal staffing level profile which should be followed for all software projects. Such profiles as a Rayleigh Curve are suggested as optimal. Chart 8 depicts characteristics of classes of projects monitored in the SEL and shows the difference in productivity and reliability for groups of projects having different staffing level profiles. Although the Rayleigh Curve may be acceptable for some projects, the SEL has found that wide variations on these characteristics still lead to a successful projects. The SEL has also found that extreme deviations may be indicative of problem software.

(Detailed information can be found in Reference 4 and 5)

Productivity for large vs. small systems (Chart 9)

The common belief by many software managers and developers is that as the size of a software system increases, its complexity increases at a higher rate than the lines of code increase. Because of this fact, it is commonly believed that in the effort equation

$$E = aI^b$$

where E = effort of person time
where I = lines of code

that the value of b must be greater than 1. The projects that the SEL has studied have been unable to verify this belief and instead have found the value of b to approximate .92 in the SEL environment. The fact that this equation is nearly linear leads to the counter intuitive point that a project of 150,000 lines of code will cost approximately 3 times as much as a 50,000 lines of code project-instead of 4 or 5 times as much as is often commonly believed.

(Further details can be found in Reference 6.)

Productivity Variation (Chart 10)

Another characteristics that the SEL has been interested in studying has been the variations in programmer productivity. Obviously one would want to increase the productivity by whatever approach found to be effective, but first we must clearly understand what the baseline characteristics of productivity are (minimum, maximum, average, difference between small and large projects, etc.); only then will we know if we have improved or not in the years to come.

As has been found by other researchers in varying environments, the productivity of different programmers can easily differ by a factor of 5 or 10 to 1. The SEL did find that there was a greater variation (from very low productivity of .5 l.o.c/hour to 10.8 l.o.c./hour) in small projects. The probable reason for this is that newer people are typically put on smaller projects and the SEL has found extreme differences in the relatively inexperienced personnel.

Reusing Code (Chart 11)

As was stated in the introduction, projects being developed in the SEL environment typically utilize approximately 30 percent old code. Although it is obviously less costly to integrate existing code into a system rather than having to generate new code, there is some cost that must be attributed to adopting the old code. The development team must test, integrate and possibly document the old code, so there is some overhead. By looking at approximately 25 projects ranging in size from 25,000 lines of code to over 100,000 total lines of code and ranging in percent of reused code from 0 percent to 70 percent, the SEL finds that by attributing a value of approximately 20 percent overhead cost to reuse code, the expenditures of the 25 projects can best be characterized. Now the SEL uses the 20 percent figure for estimating the cost of adopting existing code to a new software project.

Error Characteristics (Chart 12)

One of the other characteristics of a software environment that is of great concern to developers and managers is that of expected software reliability and that of overall software error characteristics. Before attempting to improve software reliability or before attempting to minimize the impact that software errors may have, the SEL had to first understand the error characteristics of the typical applications software in the SEL environment.

By collecting detailed error report data and through the monitoring of numerous applications projects many error characteristics have been studied.

Several pieces of information which are depicted in Chart 12 and which are based on 1381 error reports from approximately 15 projects include:

- Most errors are local to one component (subroutine or function)
- Less than 10 percent of errors were attributed to faulty requirements
- A great percent of errors (48 percent) were estimated to be trivial to correct (less than 1 hour)
- A very low percent of errors (7 percent) were estimated to be a major effort to fix (greater than 3 days)

(Further statistics and more detailed explanations can be found in References 7 and 8).

Models (Charts 11 through 16)

A second set of studies that the SEL has actively pursued is that of evaluating, reviewing, and developing software models. This includes resource models, reliability models as well as complexity metrics.

Measures for Software (Chart 14)

The SEL has attempted to utilize various available software metrics to characterize the software products generated. Such metrics as the McCabe Cyclomatic Complexity, Halstead Length, and lines of code were only a few of the measures that were reviewed.

It is commonly believed that the size of a component or the complexity of a component will be directly correlated to the reliability of that component. One set of studies performed in the SEL attempted to verify this belief. By taking over 650 modules which had very detailed records of error data, the SEL compared the correlations of 4 characteristics of the components. The characteristic included total lines of code, executable lines of code, Cyclomatic Complexity and Halstead Length. The resultant correlations are dipicted in Chart 14; which shows a very high direct correlation for the 4 measures.

A second study was performed where the error rate of each of the components was plotted against size as well as against Cyclomatic Complexity. The SEL expected to show that larger components have higher error rates than smaller components and that components of higher complexity rating had higher error rates. The plots on Chart 11 show that the results were counter-intuitive. The SEL has been unable to verify that larger or more complex components indeed have higher error rates.

Cost Models (Chart 15)

In addition to the studies made pertaining to various measures for software, the SEL has also utilized the cost data collected from the many projects to calibrate and evaluate various available resource estimation models. No attempt was intended to qualify one model as being any better than another. The objective of the studies was to better understand the sensitivities of the various models and to determine which models seemed to characterize the SEL software development environment most consistently.

In studying these resource models, 9 projects which were somewhat similar in size were used as experimental projects. Each of the models was fed complete and accurate data from the SEL data base and each was calibrated with nominal sets of projects as completely as the experimenters could. Summary results, which are given in Chart 15, indicate that, occasionally, some models can accurately predict effort required for a software project. The SEL has

reiterated what many other software developers and managers claim. Cost models should never be used as a sole source of estimation. The user must have access to experienced personnel for estimating and
must also have access to a corporate memory which can be used to calibrate and reinforce someones estimate of cost. Resource models can only be used as a supplemental tool to reinforce ones estimate or to flag possible inconsistencies.

More detailed information on the SEL studies can be found in Reference 1, 9, 10, 5

Reliability Models (Chart 16)

Another type of model that the SEL has spent some efforts in understanding and calibrating is the reliability model. Although numerous approaches have been suggested as to just how one best predicts the level of error proneness that software may have, the SEL has only performed any extended studies on one model-that which is attributed to John Musa. The model is a maximum likelihood method and the SEL attempted to apply detailed fault reports from 2 separate projects to the model in an attempt to determine if the model could accurately predict remaining faults in the software.

Chart 16 indicates that one of the experiments was quite successful and one of the experiments was not successful. It should be noted that during and after these experiments, John Musa reviewed the results and the data very carefully and he has pointed out some possible deficiencies in the SEL data which could possibly lead to erroneous results in this application of the reliability model. One such piece of data is the granularity with which computer CPU time is recorded between reported faults. The SEL data is not as accurate as the model calls for.

The charts show that for experiment 1, the model quite accurately predicted a level of reliability after approximately 1/2 of the total uncovered faults were reported. The chart also shows that for experiment 2, the model was still predicting a very high number of errors to be still in the software, when in fact a minimal set were ever uncovered during the several years of operation for that system.

More detailed discussions can be found in Reference 1 and 11.

Methodologies (Charts 17 through 20)

As was mentioned earlier, one of the major objectives of the SEL has been to measure the effectiveness of various software development methodologies. The SEL has utilized selected development approaches in different applications software tasks and then has analyzed the process and product to study the relative impact of the approach. A summary of some of the results of the experimentation process is presented here.

Use of An Independent Verification and Validation Team (Chart 18)

Many software managers, developers and organizations have advocated the usage of an independent IV&V team during the software development process. The major advantage of following such an approach, it is claimed, will be the improvement in software reliability, quality, visibility, but not necessarily an improvement in overall software productivity.

In an attempt to evaluate the impact that the usage of an IV&V team may have on the SEL environment, 3 candidate projects were selected to utilize the methodology of an IV&V. Two of the projects were very typical flight dynamics systems, each containing over 50,000 lines of code while the third was a smaller flight dynamics project comprised of about 10,000 lines of code. In addition to the IV&V approach being applied to the projects, the development teams utilized the commonly followed standards and approaches normally used by development efforts within the SEL environment.

The projects lasted approximately 18 months, and the IV&V effort was active for the entire duration of the project. The size of the IV&V effort was about 18 percent of the effort of each of the large development efforts. A series of measures was defined near the beginning of the experiment by the SEL. These measures would be used to determine whether or not the application of the IV&V approach was cost effective in the SEL environment.

A summary of some of the measures is depicted in Chart 18. The results here indicate:

- o total cost of the project increased-as expected
- o productivity of the development teams (not counting the cost of IV&V) was among the lowest of any previous SEL monitored project.
- o rates of uncovering errors found earlier in the development cycle was better
- o cost rate to fix all discovered errors was no less than in any other SEL projects
- o reliability of the software (error rate during acceptance testing and during maintenance and operations) was no different than other SEL projects

The conclusion of the SEL, based on these 3 experiments, was that the IV&V methodology was not an effective approach in this SEL environment.

(A more detailed description can be found in Reference 12).

Effects of MPP on Software Development (Chart 19)

In an attempt to determine if the utilization of Modern Programming Practices (MPP) has any impact (either favorable or unfavorable) on the development of software, a set of 10 fairly large (between 50,000 l.o.c. and 120,000 l.o.c.), and fairly similar projects (same development environment, same type of requirements, same time constraints) was closely examined. These projects all had been developed in the SEL environment where detailed information was extracted from the projects weekly and where each project had a different level of MPP enforced during the development process.

The MPP's ranged from various design approaches (such as PDL, Design Walk Throughs, etc.) to code and test methodologies (such as structured code, code reading, etc.), to various integration and system testing approaches. All of the possible MPP's were rated and scaled as to the level to which the practice was followed for each project (the rating was done by the SEL researchers, not by the software developers). The only purpose of this exercise was to depict trends and not to prove that any one single practice was more effective by itself than any other.

The level to which MPP's were utilized were plotted against productivity and against error rate. Chart 19 indicates that the application of the MPP has favorably affected productivity by about 15 percent for these experiments. The results of software reliability vs MPP is very questionable. The SEL is still continuing analysis of additional data. The chart shown is obviously very inconclusive.

(More details of this effort can be found in Reference 13).

Subjective Summary of Effective Practices (Chart 20)

The previous chart indicated that productivity can be improved by an appreciable amount if certain, select practices are applied to the software development process. One obviously next would ask, which practices are the most effective? The SEL has been attempting to analyze the available data from the 40 experiments it has conducted to answer this very question. As was stated earlier, the SEL feels that these types of experiments can only depict trends and cannot accurately isolate one practice as measurable on its own. Whether or not this can be done, or whether one should ever attempt it is questionable. Most software development methodologies represent an integrated set of practices that only are effective when they are applied in a combined, uniform fashion. Most practices do not make sense, or at least cannot be effective as a stand alone approach.

A summary of the trends that the SEL has discovered for specific experiments conducted is represented in Chart 20. This chart is a combination of experimental results and subjective information from the experimenters and users and should only be viewed as depicting trends in various approaches. No numerical value of impact can realistically be assigned to the individual practices tested. It seems that practices such as PDL, code reading and librarian have proved most beneficial while such techniques as automated flow charters, requirements languages and the axriomatic design approach have been unsuccessful in the SEL.

Cost of Data Collection (Chart 21)

The SEL has been in existence for about 7 years and has been collecting detailed software development data for over 6 years. Numerous experiments have been conducted in an attempt to understand and measure various methodologies for developing software. In support of these efforts, one of the most critical and difficult elements of the entire experimentation process is that of data collection.

The data collection process is time consuming, frustrating, sometimes unrewarding, and most assurably is expensive. Chart 21 shows the overhead cost that the SEL has experienced over the past 6 years. To accurately collect data from the development tasks, the SEL finds that there is a 3 to 7 percent overhead price on the development effort. To process the data that has been collected (verification, encoding, data entry, storage, etc.), the SEL has spent approximately an additional 10 to 12 percent of the development effort. Finally, the SEL experiences indicate that one can spend up to an additional 25 percent of the development effort to perform the detailed analysis of the data that has been collected. This includes support before, during and after the experiments in defining the data to be collected, monitoring the development data and effort, formulating hypothesis and performing analysis of the completed experiments. The product of the analysis consists of papers, reports, and documents.

(Detailed information on cost can be found in Reference 2).

Summary (Chart 22)

In summary, the SEL has had much experience with the data collection process and with the experimentation process. Many of its attempts have been rewarding and many have been fruitless, but the SEL feels attempts to assess approaches to software have to be conducted if we are ever to evolve to a more productive approach to developing software.

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. SEL, 81-101, Guide to Data Collection, V. E. Church, D. N. Card, F. E. McGarry, et. al., August 1982

3. SEL, 81-102, Software Engineering Laboratory (SEL) Data Base Organization and User's Guide, D. C. Wyckoff, G. Page, F. E. McGarry, et. al., March 1983

4. 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

5. 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

6. Basili, V. R., and K. Freburger, 'Programming Measurement and Estimation in the Software Engineering Laboratory', Journal of Systems and Software, February 1981, Volume 2, No. 1

7. SEL 81-011, Evaluating Software Development by Analysis of Change Data, D. M. Weiss, November 1981

8. Basili, V. R., and B. T. Perricone, Software Errors and Complexity: An Empirical Investigation, University of Maryland, Technical Report TR-1195, August 1982

9. SEL 80-007, An Appraisel of Selected Cost/Resource Estimation Models for Software Systems, J. F. Cook, F. E. McGarry, December 1980

10. Basili, V. R., 'Software Engineering Laboratory Relationships for Programming Measurment and Estimation', University of Maryland, Technical Memorandum, October 1979

11. SEL 80-005, A Study of the Musa Reliability Model, A. M. Miller, November 1980

12. SEL 81-110, Performance and Evaluation of an Independent Software Verification and Integration Process, G. Page. and F. McGarry, September 1982

13. SEL 82-001, Evaluation of Management Measures of Software Development, D. Card, G. Page, F. McGarry, September 1982

MEASURING SOFTWARE DEVELOPMENT TECHNOLOGY

OR

SHOULD PROGRAMMERS DO IT TOP DOWN ?

F. McGarry
NASA/GSFC

CHART 1

SEL APPROACH TO SOFTWARE TECHNOLOGY ASSESSMENT

SOFTWARE EXPERIMENTS IN PRODUCTION ENVIRONMENT: NASA APPLICATIONS

- **DEVELOP PROFILE OF ENVIRONMENT (SCREENING)**
  - EXTRACT DETAILED DEVELOPMENT DATA
  - DETERMINE CHARACTERISTICS OF DEVELOPMENT PROCESS
- **EXPERIMENT WITH PROPOSED TECHNOLOGIES (CONTROLLED)**
  - APPLY VARIOUS TECHNOLOGIES (METHODS, MODELS, AND TOOLS) TO APPLICATIONS PROGRAMS
  - EXTRACT DETAILED DEVELOPMENT DATA
- **MEASURE IMPACT AND/OR ASSESS TECHNOLOGIES**
  - DEFINE MEASURES FOR EVALUATION
  - COMPARE EFFECTS OF USING OR NOT USING APPROACHES IN QUESTION (SIMILAR PROJECTS)
  - DETERMINE EFFECTIVENESS OF TECHNOLOGIES IN QUESTION (WHICH ONES HELP AND BY HOW MUCH)

F. McGarry
NASA/GSFC

SOFTWARE ENVIRONMENT

DEVELOPMENT LANGUAGE	FORTRAN (15% MACROS)
SOFTWARE TYPE	SCIENTIFIC, GROUND-BASED INTERACTIVE, NEAR-REAL-TIME
SIZE	TYPICALLY~60,000 SLOC (2,000 TO 110,000)
DEVELOPMENT TIME	16 TO 24 MONTHS (START DESIGN TO START OPERATIONS)
STAFFING	6 TO 14 PERSONS
DEVELOPMENT SYSTEM	IBM S/360 (PRIMARILY) VAX-11/780 PDP-11/70

EXPERIMENTS WITHIN THE SEL 1977 THROUGH 1982 BASIS FOR SUMMARY INFORMATION AND CONCLUSIONS

LABORATORY EXPERIMENTS	46 PROJECTS
INFORMATION MONITORED	1.8 MILLION SLOC
PROGRAMMERS/MANAGERS REPRESENTED	150 PEOPLE
DATA EXTRACTED	20,000 FORMS
METHODOLOGIES APPLIED	200 QUALIFYING PARAMETERS AND VARIOUS MODELS, TOOLS, AND METHODOLOGIES

334-PAG-(2*)

AREAS OF DISCUSSION

- PROFILES
- MODELS
- METHODOLOGIES

334-PAG-(2*)

F. McGarry
NASA/GSFC

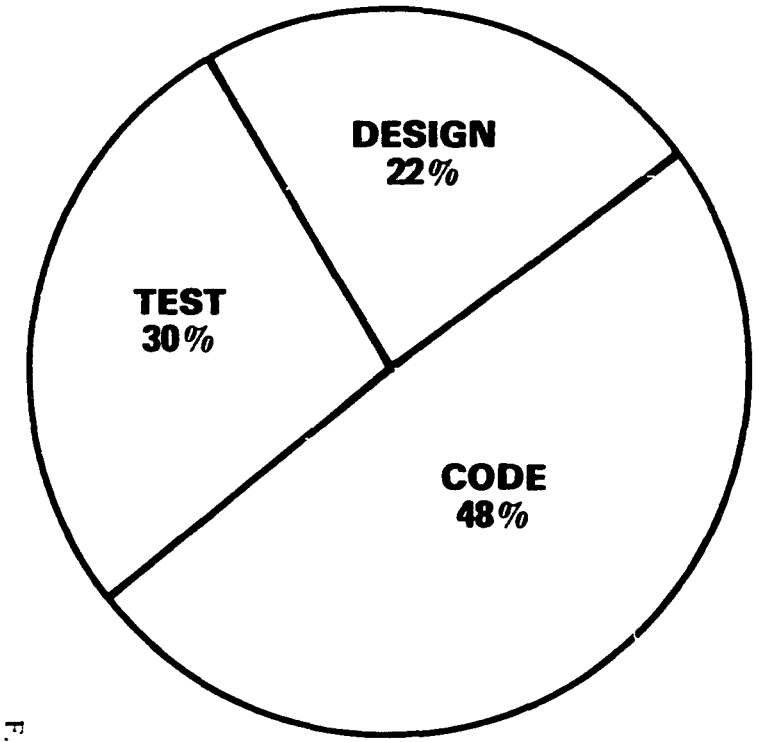
CHART 5

PROFILES

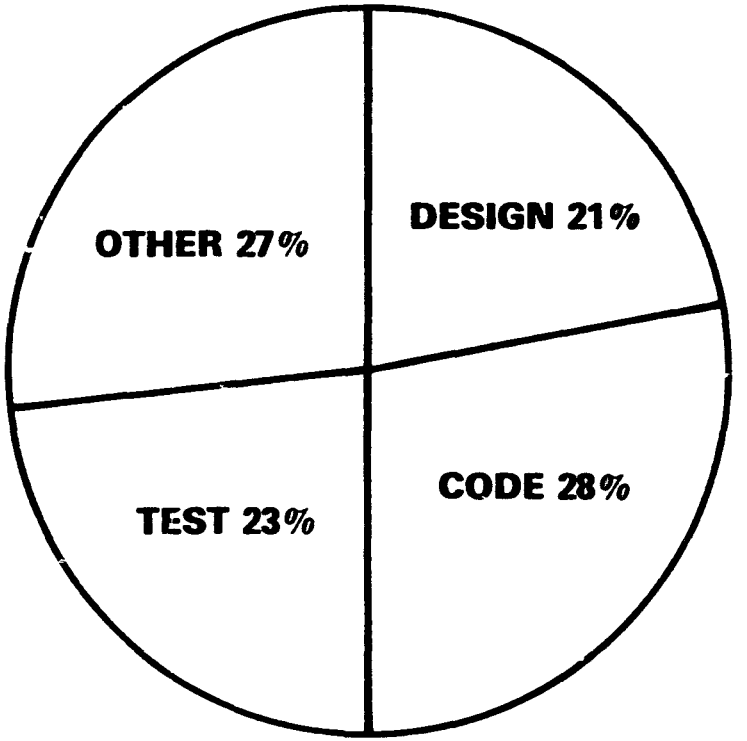
334-PAG-(2*)

WHERE DO PROGRAMMERS SPEND THEIR TIME?

DATE DEPENDENT



PROGRAMMER REPORTING

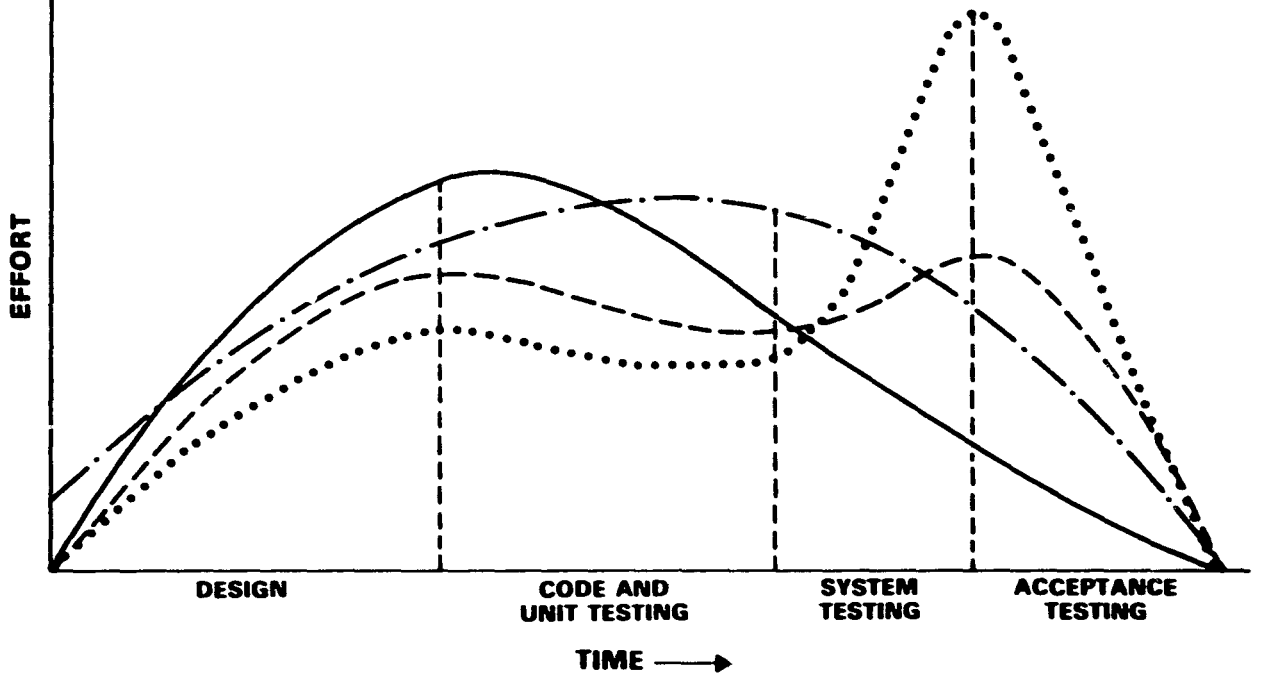


334-PAG-(2*)

F. McGarry
NASA/GSFC

CHART 7

PROFILES OF DEVELOPMENT RESOURCES
HOW MANY ROADS TO COMPLETION?



PROFILE	PRODUCTIVITY (SLOC/HOUR)	RELIABILITY (ERRORS/K SLOC)
———	RAYLEIGH CURVE	–
—·—	4.4–4.6	UP TO 2
– – –	2.7–4.7	UP TO 2
•••••••	2.7–2.9	UP TO 2

- RELATIONSHIP BETWEEN PROFILE AND PRODUCTIVITY
- NO RELATIONSHIP BETWEEN PROFILE AND RELIABILITY

CHART 8

ARE LARGE PROGRAMS HARDER TO BUILD THAN SMALL ONES?

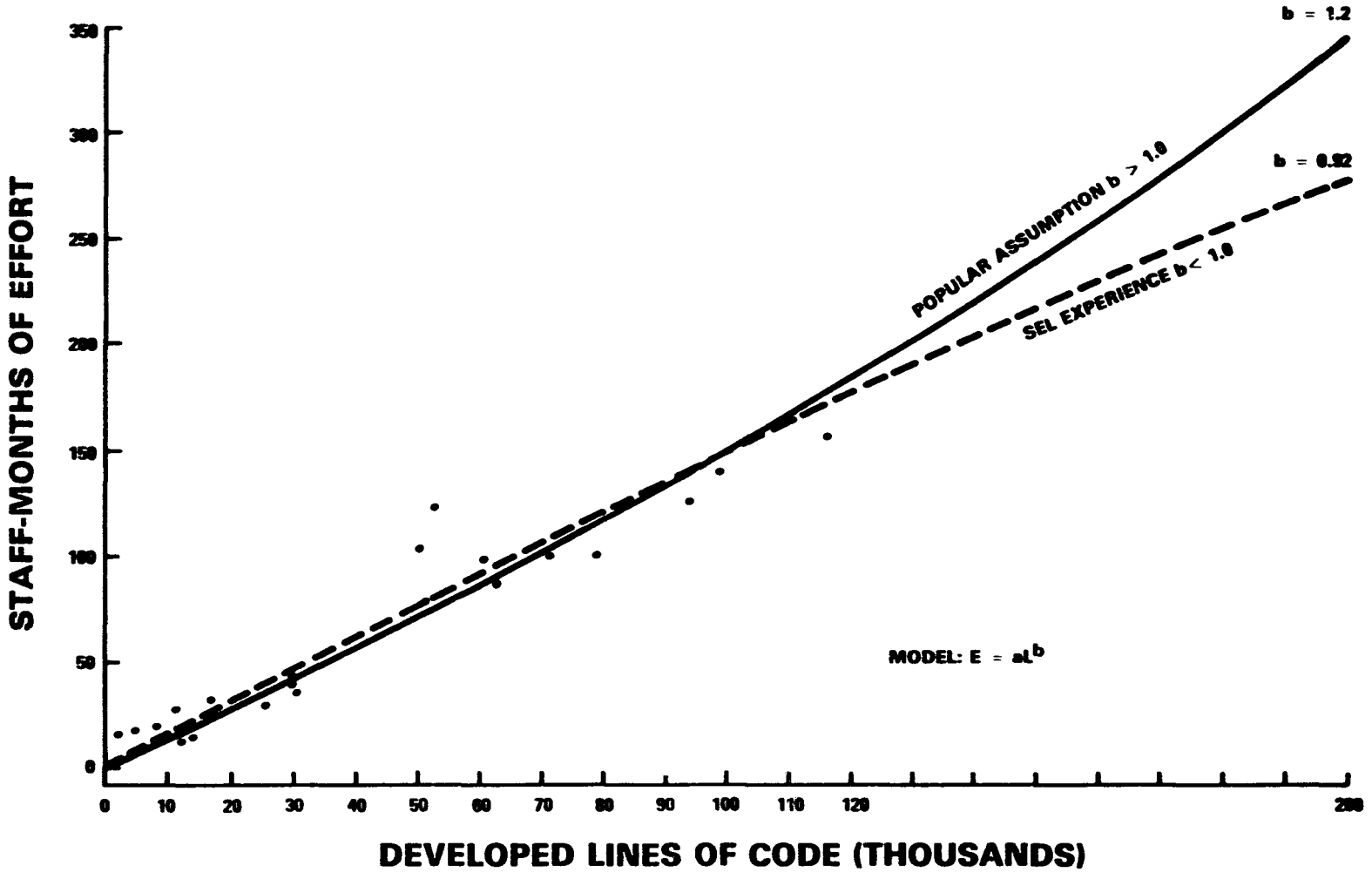
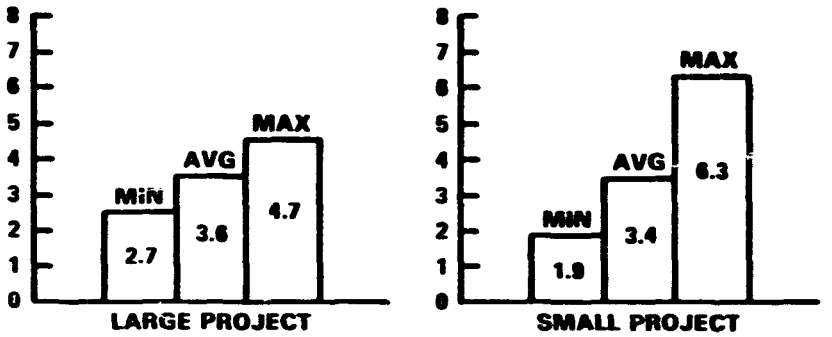


CHART 9

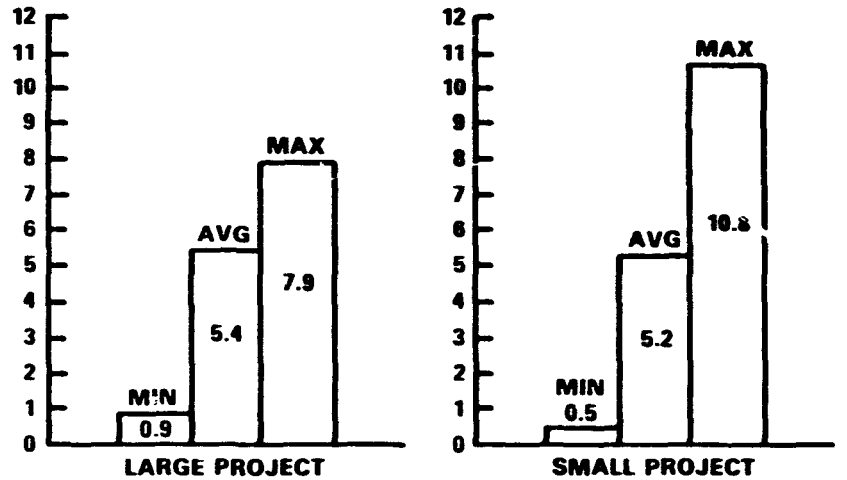
334-PAG-(2c*)

PRODUCTIVITY VARIATION (SLOC/HOUR)[1]

BY PROJECT (ALL CHARGES)



BY PERSON (PROGRAMMER ONLY)



PEOPLE ARE THE MOST IMPORTANT METHODOLOGY

[1] A LARGE PROJECT IS GREATER THAN 20K SLOC.

224-PAG-(2a*1)

ASSESSING REUSED CODE

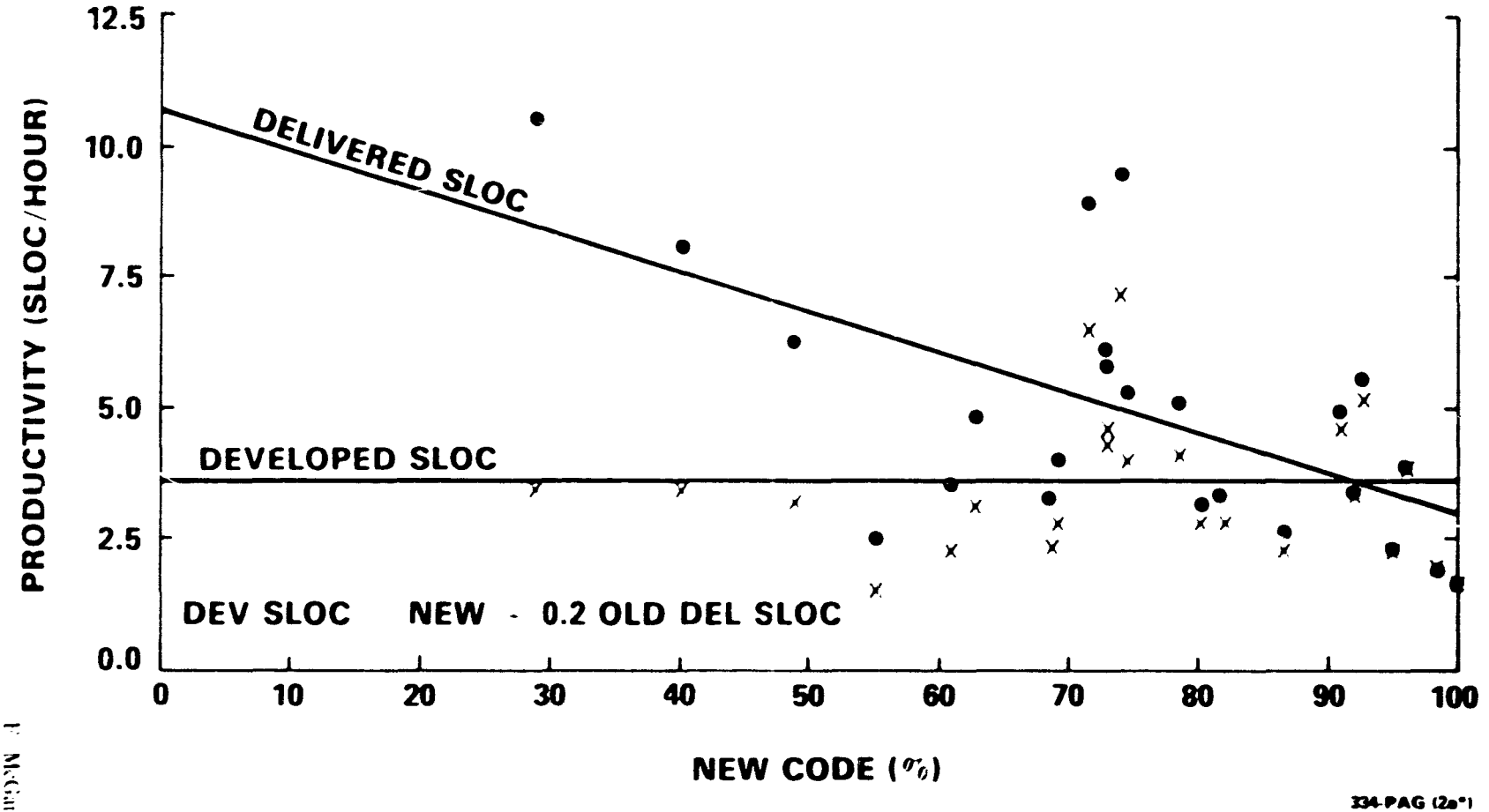
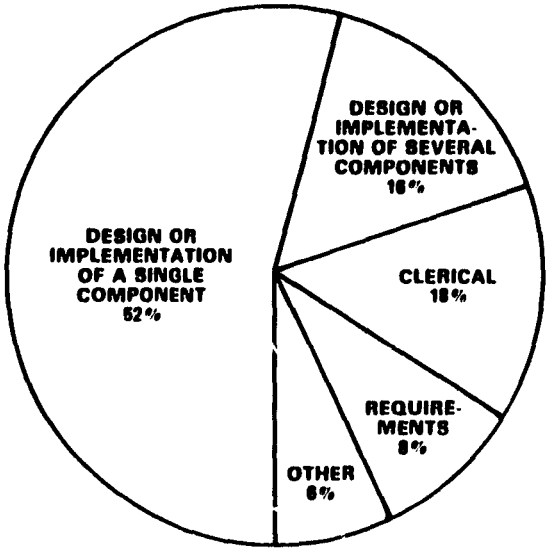


CHART 11

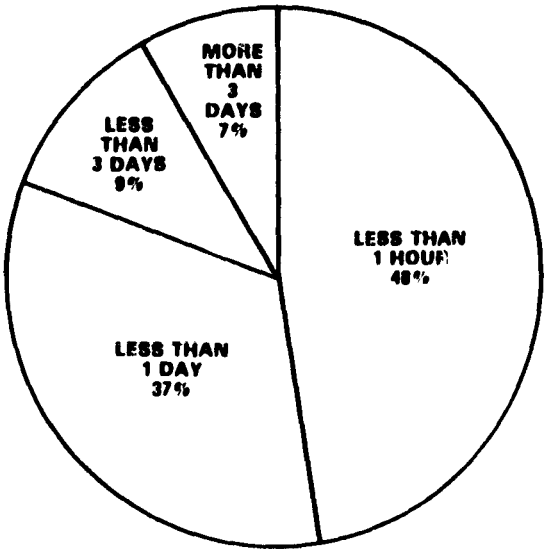
ERROR CHARACTERISTICS
(MEASURED DURING IMPLEMENTATION)

TYPES OF ERRORS



SAMPLE OF 1381 REPORTS

EFFORT TO CORRECT



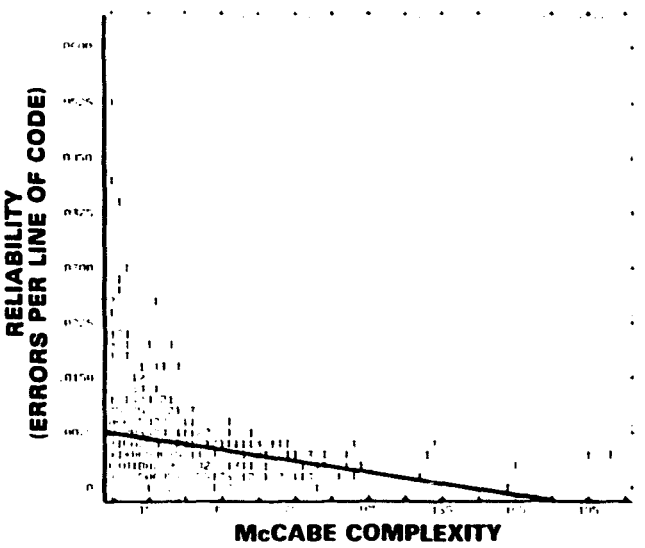
- **MOST ERRORS ARE EASY TO CORRECT**
- **SEVERAL-COMPONENT ERRORS ARE LESS THAN EXPECTED**
- **REQUIREMENTS ERRORS ARE LESS THAN EXPECTED**

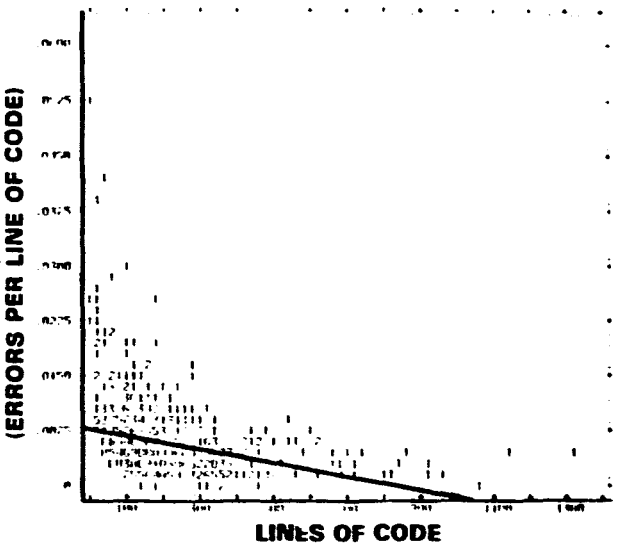
MODELS

334-PAG-(Zb*)

CHART 13

SOFTWARE MEASURES IN THE SEL





	CORRELATIONS			
	TOTAL LINES	EXECUTABLE LINES	McCAPE COMPLEXITY	HALSTEAD LENGTH
HALSTEAD LENGTH	0.85	0.91	0.91	1.00
McCABE COMPLEXITY	0.81	0.87	1.00	
EXECUTABLE LINES	0.84	1.00		
TOTAL LINES	1.00			

SAMPLE OF 688 MODULES

334-PAG-(2c*)

ORIGINAL PAGE IS
OF POOR QUALITY

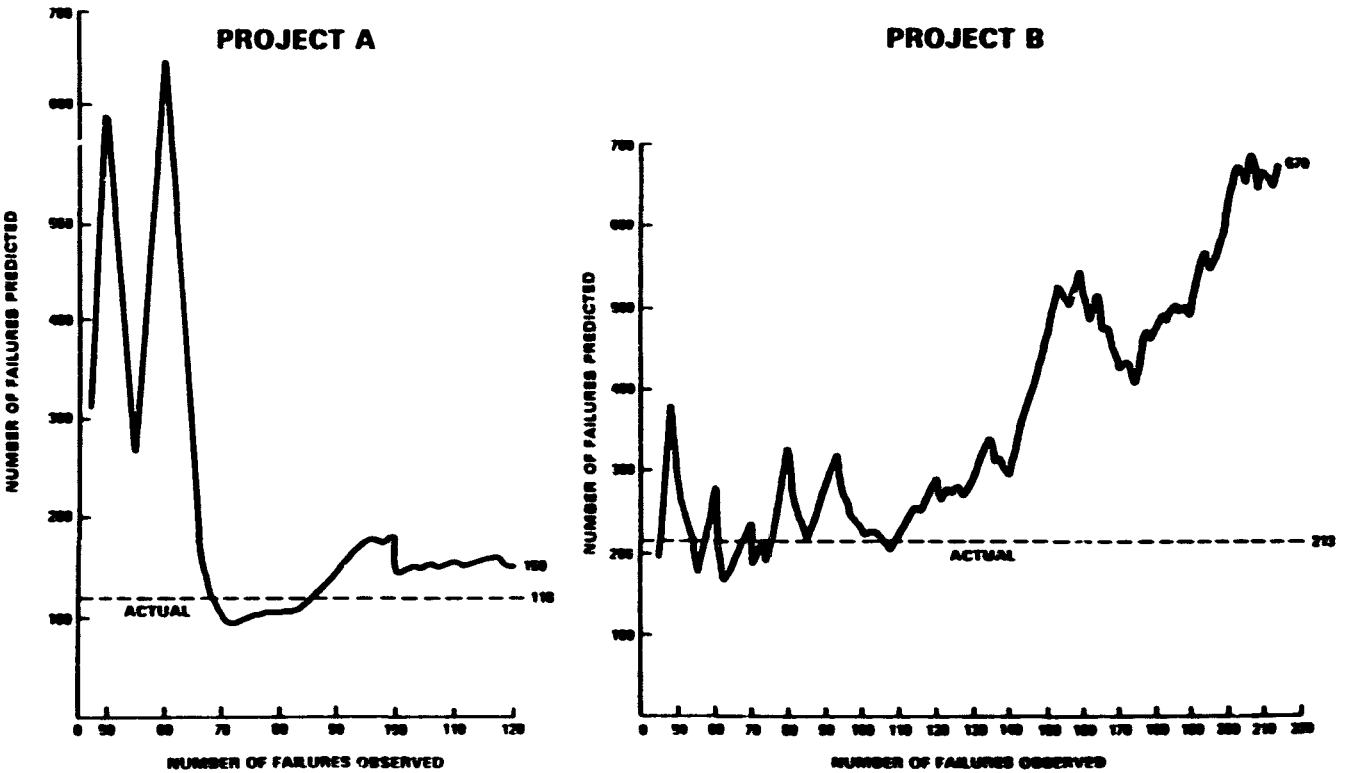
COMPARISON OF COST MODELS

PROJECT	ACTUAL EFFORT (MM)	PERCENTAGE OF ERROR IN PREDICTION				
		DOTY	PRICE S3	TECOLOTE	SEL	COCOMO
1	79	+65	+8	−4	−6	—
2	96	+30	+6	−25	−22	+1
3	40	+65	+6	−8	+93	—
5	98	+74	0	+3	−2	+2
6	116	+123	+36	+35	−3	—
7	91	+52	+14	−12	−14	—
8	99	+127	+7	+36	+14	+53
9	106	—	—	—	−24	+16

SOMETIMES, SOME MODELS WORK WELL

334-PAG-(2b*)

PREDICTING RELIABILITY
(MUSA MAXIMUM LIKELIHOOD METHOD)



WE DON'T KNOW ENOUGH ABOUT RELIABILITY MODELS

CHART 16

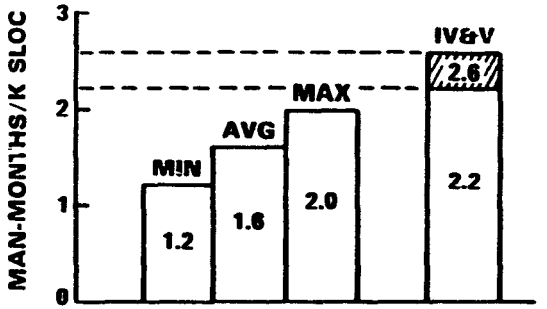
METHODOLOGIES

F. McGarry
NASA/GSFC

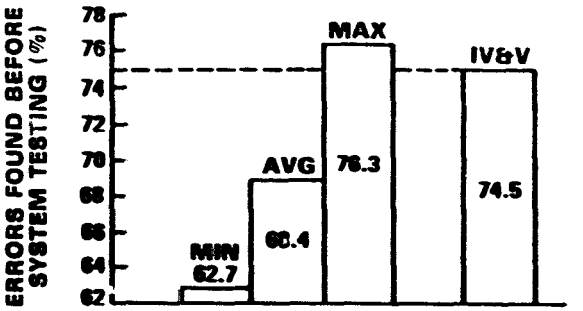
CHART 17

A LOOK AT IV&V METHODOLOGY

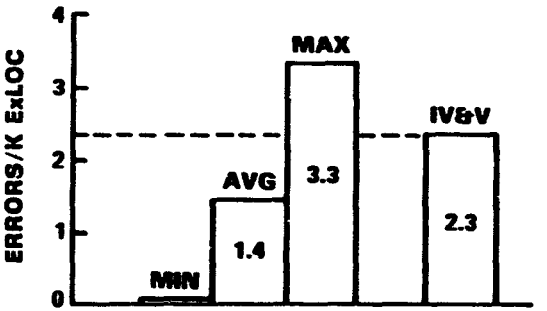
(BASED ON RESULTS FROM 3 EXPERIMENTS)



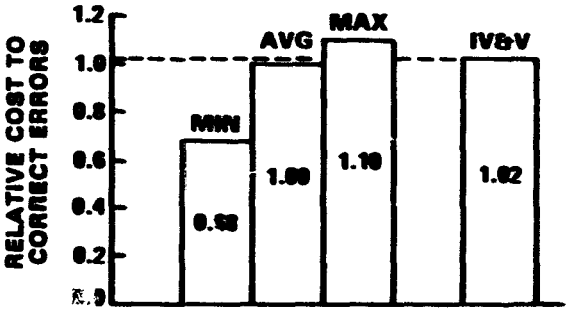
- COST INCREASED



- MORE ERRORS FOUND EARLY



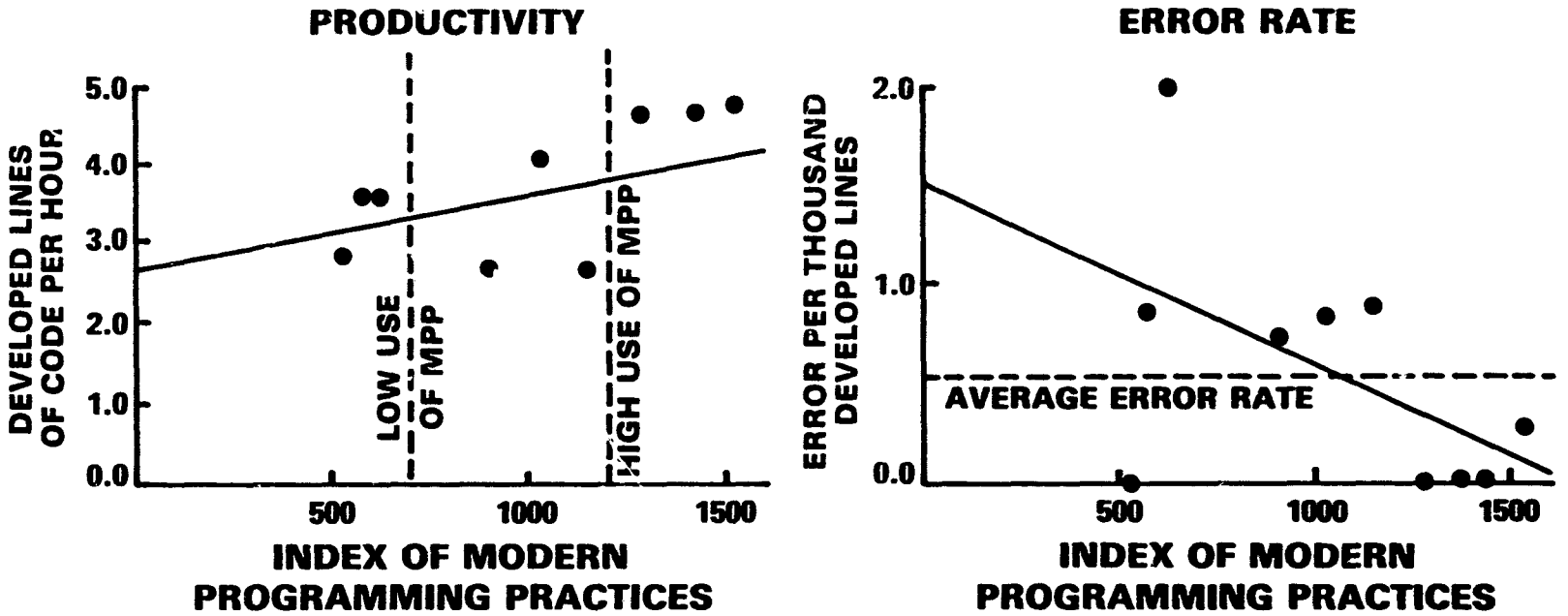
- RELIABILITY NOT IMPROVED



- ERROR CORRECTION COST NOT DIFFERENT

- **IF YOU MULTIPLY ERRORS FOUND EARLY BY A LATENCY FACTOR, IV&V LOOKS GOOD**
- **IF YOU EXAMINE ALL MEASURES, IV&V LOOKS BAD**

EFFECTS OF MPP ON SEL SOFTWARE DEVELOPMENT



- PRODUCTIVITY IS ABOUT 15 PERCENT HIGHER
- RELIABILITY IS HIGHLY VARIABLE

334-PAG-(2d*)

F. McGarry
NASA/GSFC

WHAT HAS BEEN SUCCESSFUL IN OUR ENVIRONMENT?

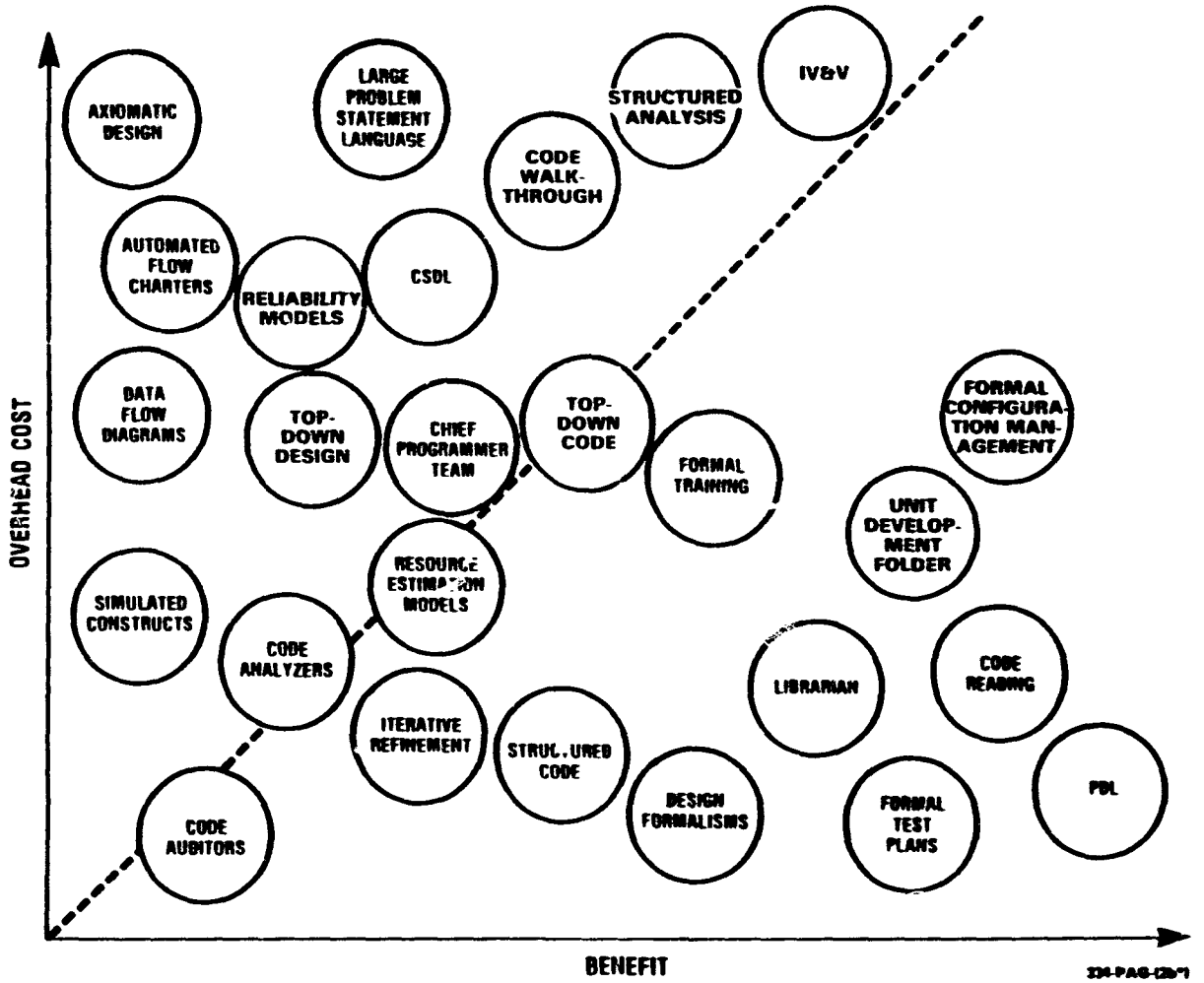


CHART 20

COST OF DATA COLLECTION
(AS A PERCENTAGE OF TASKS BEING MEASURED)

	SEL EXPERIENCES
OVERHEAD TO TASKS (EXPERIMENTS)	**3—7%**
● FORMS	
● MEETINGS	
● TRAINING	
● INTERVIEWS	
● COST OF USING TOOLS	
DATA PROCESSING	**10—12%**
● COLLECTING/VALIDATING FORMS	
● ARCHIVING/ENTERING DATA	
● DATA MANAGEMENT AND REPORTING	
ANALYSIS OF INFORMATION	**UP TO 25%**
● DESIGNING EXPERIMENTS	
● EVALUATING EXPERIMENTS	
● DEFINING ANALYSIS TOOLS	

SUMMARY

- DATA COLLECTION IS EXPENSIVE — BUT VERY, VERY IMPORTANT
- WE MUST UNDERSTAND WHERE WE ARE BEFORE HEADING SOMEWHERE ELSE
- EXPERIMENTATION WILL PAY FOR ITSELF (TRY SOMETHING NEW)
- MPP CAN FAVORABLY IMPACT PRODUCTIVITY AND RELIABILITY
- SOME METHODOLOGIES BUY YOU NOTHING (OR EVEN WORSE)
- MODELS MUST BE UTILIZED WITH GREAT CARE

CHART 22

PANEL #1

THE SOFTWARE ENGINEERING LABORATORY (SEL)

V. Basili, University of Maryland
A. Goel, Syracuse University
M. Zelkowitz, University of Maryland

D2

N83 32358

SOFTWARE ERRORS AND COMPLEXITY: AN EMPIRICAL INVESTIGATION

Victor R. Basili and Barry T. Perricone

Department of Computer Science

University of Maryland

College Park, Md.

1982

ABSTRACT

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

1.0 INTRODUCTION

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

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

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

1.1 DESCRIPTION OF THE ENVIRONMENT

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

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

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

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

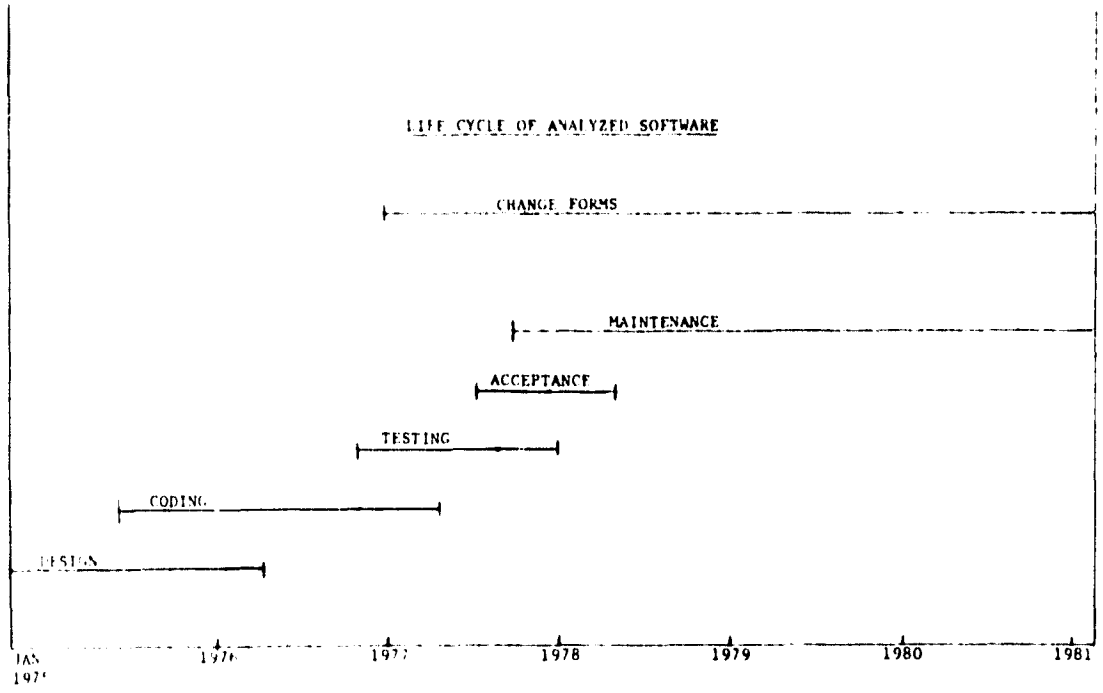


Figure 1

1.2 TERMS

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

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

There are two types of modules referred to within this paper. The first type is denoted as modified. These are

modules which were developed for previous software projects and then modified to meet the requirements of the new project. The second type is referred to as new. These are modules which were developed specifically for the software project under analyses.

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

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

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

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

Table 1

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

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

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

first type of quantification described.

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

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

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

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

2.0 BASIC DATA

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

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

the change or error, the number of components changed, the type of change or error, and the effort needed to correct the error.

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

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

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

3.0 RELATIONSHIPS DERIVED FROM DATA

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

3.1 CHANGE DISTRIBUTION BY TYPE

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

3.2 ERROR DISTRIBUTION BY MODULES

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

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

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

Figure 2

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

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

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

Figure 3

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

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

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

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

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

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

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

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

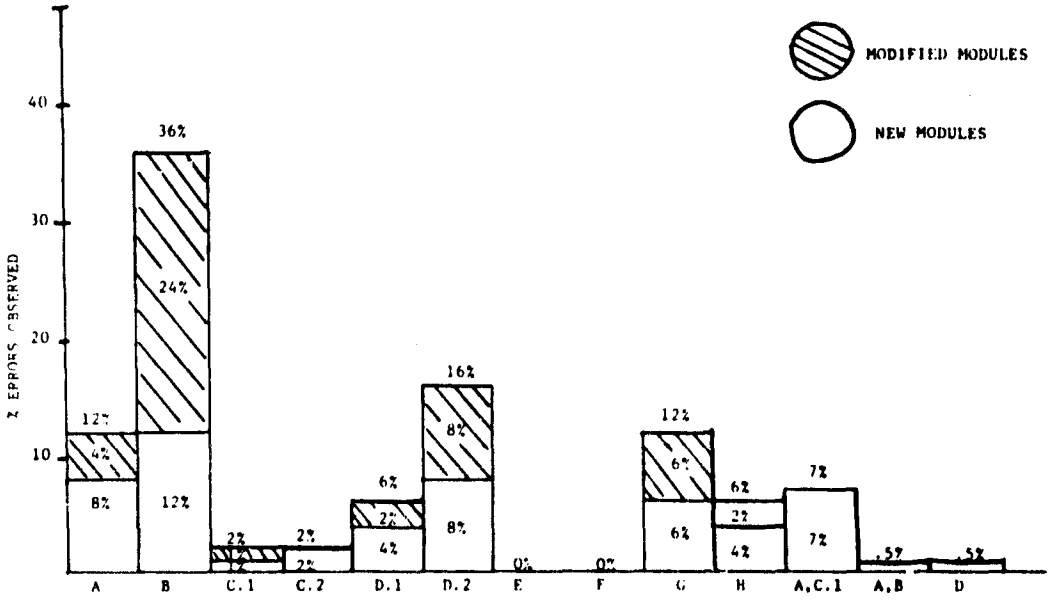
3.3 ERROR DISTRIBUTION BY TYPE

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

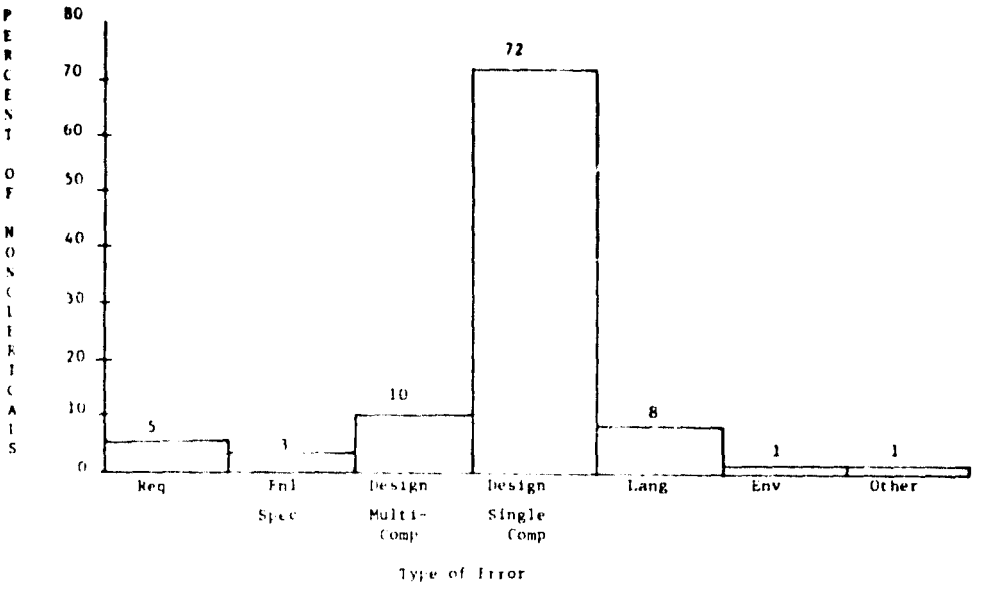
The classification for error used throughout the Software Engineering Laboratory is given below. The person identifying the error indicates the class for each error.

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

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



SOURCES OF ERRORS
Figure 4



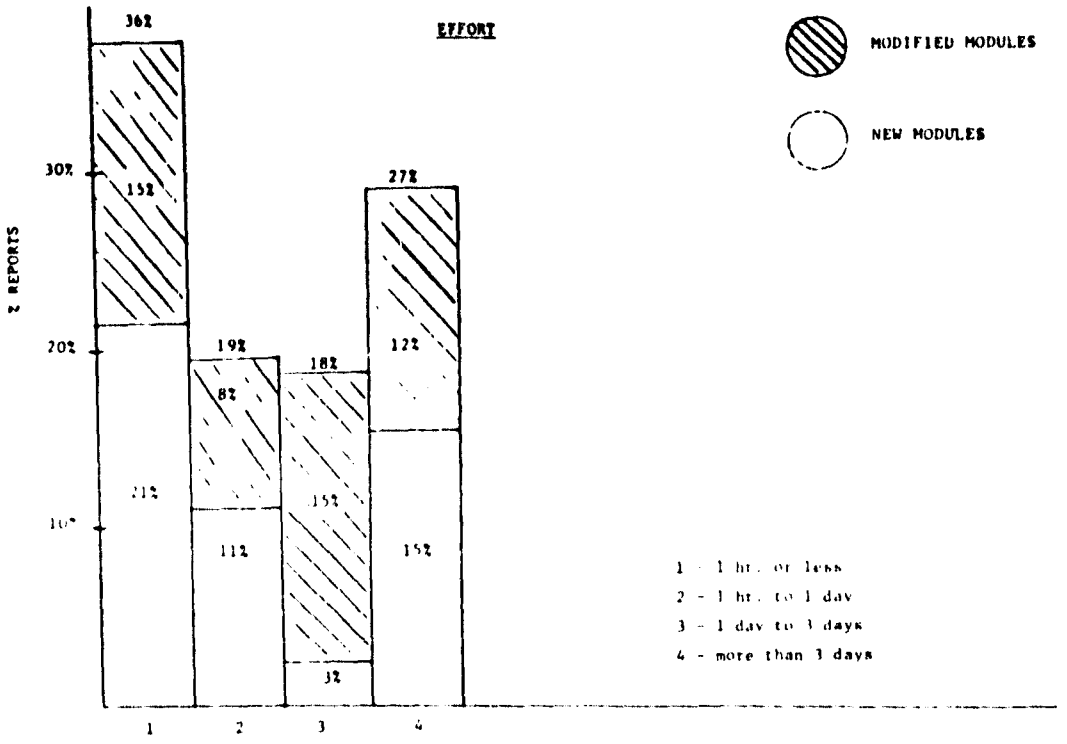
SOURCES OF ERROR ON OTHER PROJECTS
Figure 5

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

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

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

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



EFFORT GRAPH
Figure 6

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

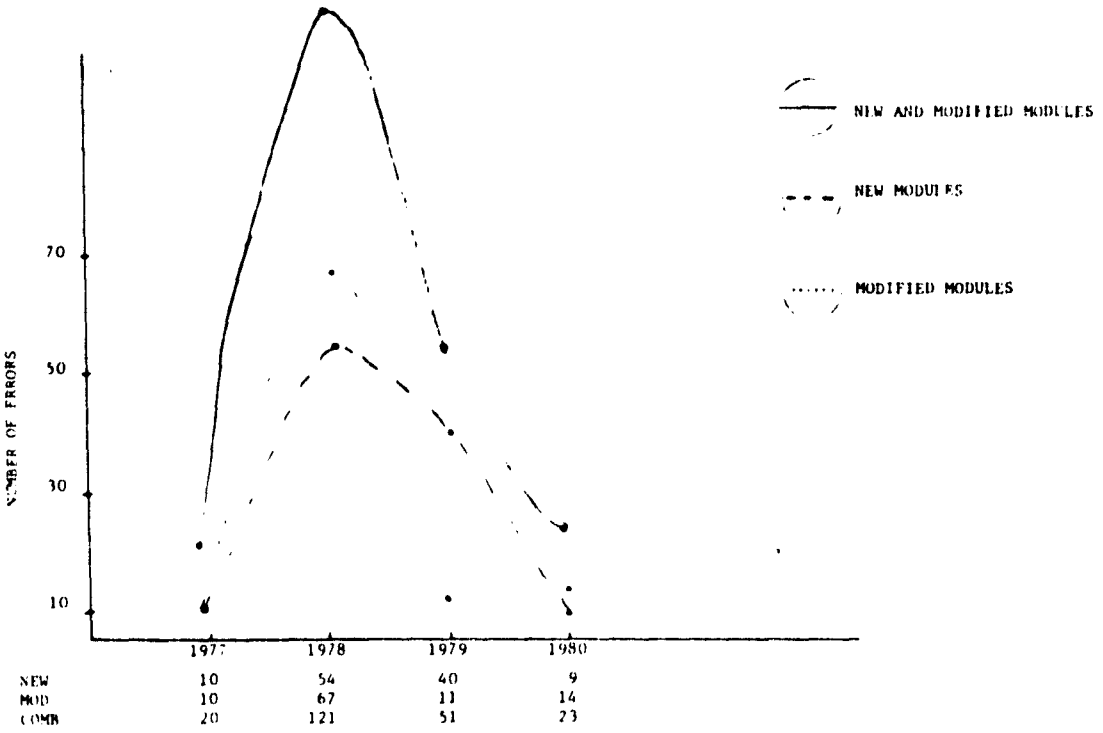
In general, these observations tend to indicate that there are disadvantages in modifying a large number of already existing modules to meet new specifications. The alternative of developing a new module might be better in some cases if there does not exist good specifications for the existing modules.

3.4 OVERALL NUMBER OF ERRORS OBSERVED

Figure 7 displays the number of errors observed in both new and modified modules. The curve representing total

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

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



NUMBER OF ERRORS OCCURRING IN MODULES
Figure 7

3.5 ABSTRACT ERROR TYPES

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

analyst may have interpreted the categories differently.

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

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

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

description given on the change report form.

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

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

ABSTRACT CLASSIFICATION OF ERRORS
Figure 8

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

3.6 MODULE SIZE AND ERROR OCCURRENCE

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

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

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

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

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

3.7 MODULE COMPLEXITY

Cyclomatic complexity [5] (number of decisions + 1) was correlated with module size. This was done in order to

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

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

AVERAGE CYCLOMATIC COMPLEXITY FOR ALL MODULES
Table 5

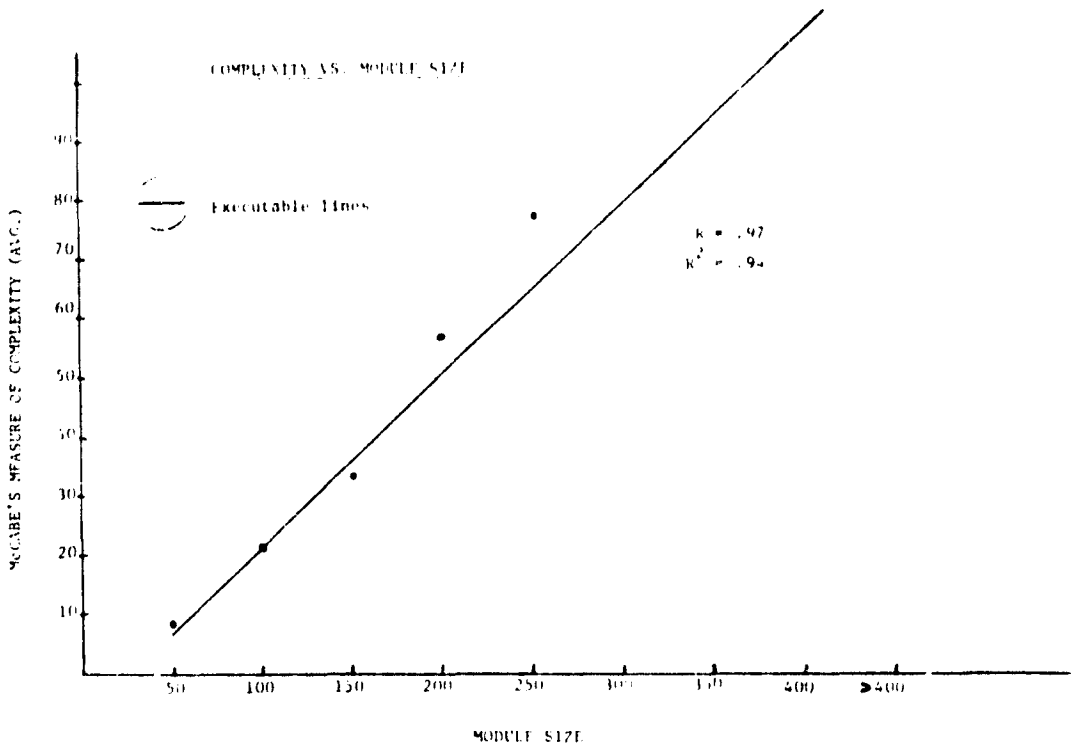


Figure 9

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

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

COMPLEXITY AND ERROR RATE FOR ERRORED MODULES
Table 6

4.0 DATA NOT EXPLICITLY INCLUDED IN ANALYSES

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

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

5.0 CONCLUSIONS

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

Modified and new modules were shown to behave similarly except in the types of errors prevalent in each and the amount of effort required to correct an error. Both had a high percentage of interface errors, however, new modules had an equal number of errors of omission and commission and a higher percentage of control errors. Modified modules had a high percentage of errors of commission and a small percentage of errors of omission with a higher percentage of

data and initialization errors. Another difference was that modified modules appeared to be more susceptible to errors due to the misunderstanding of the specifications. Misunderstanding of a module's specifications or requirements constituted the majority of errors detected. This duplicates an earlier result of Endres which implies that more work needs to be done on the form and content of the specifications and requirements in order to enable them to be used across applications more effectively.

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

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

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

Acknowledgments

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

References

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

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

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

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

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

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

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

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

NUMBER _____

CHANGE REPORT FORM

PROJECT NAME _____ CURRENT DATE _____

SECTION A - IDENTIFICATION

REASON: Why was the change made? _____

DESCRIPTION: What change was made? _____

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

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

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

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

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

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

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

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

FOR ERROR CORRECTIONS ONLY

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

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

FOR DESIGN OR IMPLEMENTATION ERRORS ONLY

→ If the error was in design or implementation:

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

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

Change Report Form

FOR ERROR CORRECTIONS ONLY

SECTION D - VALIDATION AND REPAIR

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

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

What was the time used to isolate the cause?

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

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

Was this error related to a previous change?

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

When did the error enter the system?

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

SECTION E - ADDITIONAL INFORMATION

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

Name _____ Authorized: _____ Date: _____

Change Report Form

THE VIEWGRAPH MATERIALS

for the

V. BASILI PRESENTATION FOLLOW

SOFTWARE ERRORS AND COMPLEXITY: AN EMPIRICAL INVESTIGATION

VICTOR R. BASILI
BARRY T. PERRICONE

UNIVERSITY OF MARYLAND

STUDY OVERVIEW

STUDY THE ERRORS COMMITTED IN DEVELOPING SOFTWARE

REVIEW THE RESULTS IN LIGHT OF THOSE FROM OTHER STUDIES

ANALYZE THE RELATIONSHIP BETWEEN ERRORS AND COMPLEXITY

PROJECT BACKGROUND

GENERAL PURPOSE PROGRAM FOR SATELLITE PLANNING STUDIES

SIZE: 90K SOURCE LINE/517 CODE SEGMENTS

370 FORTRAN SUBROUTINES/36 ASSEMBLY SEGMENTS/111 COMMON MODULES, BLOCK DATA, UTILITY ROUTINES

MODIFIED MODULES - ADOPTED FROM A PREVIOUS SYSTEM (72%)

NEW MODULES - DEVELOPED SPECIFICALLY FOR THIS SYSTEM

REQUIREMENTS FOR THE SYSTEM KEPT GROWING AND CHANGING OVER THE LIFE CYCLE

ERRORS: TWO DEFINITIONS - TEXTUAL (215) AND CONCEPTUAL (155)

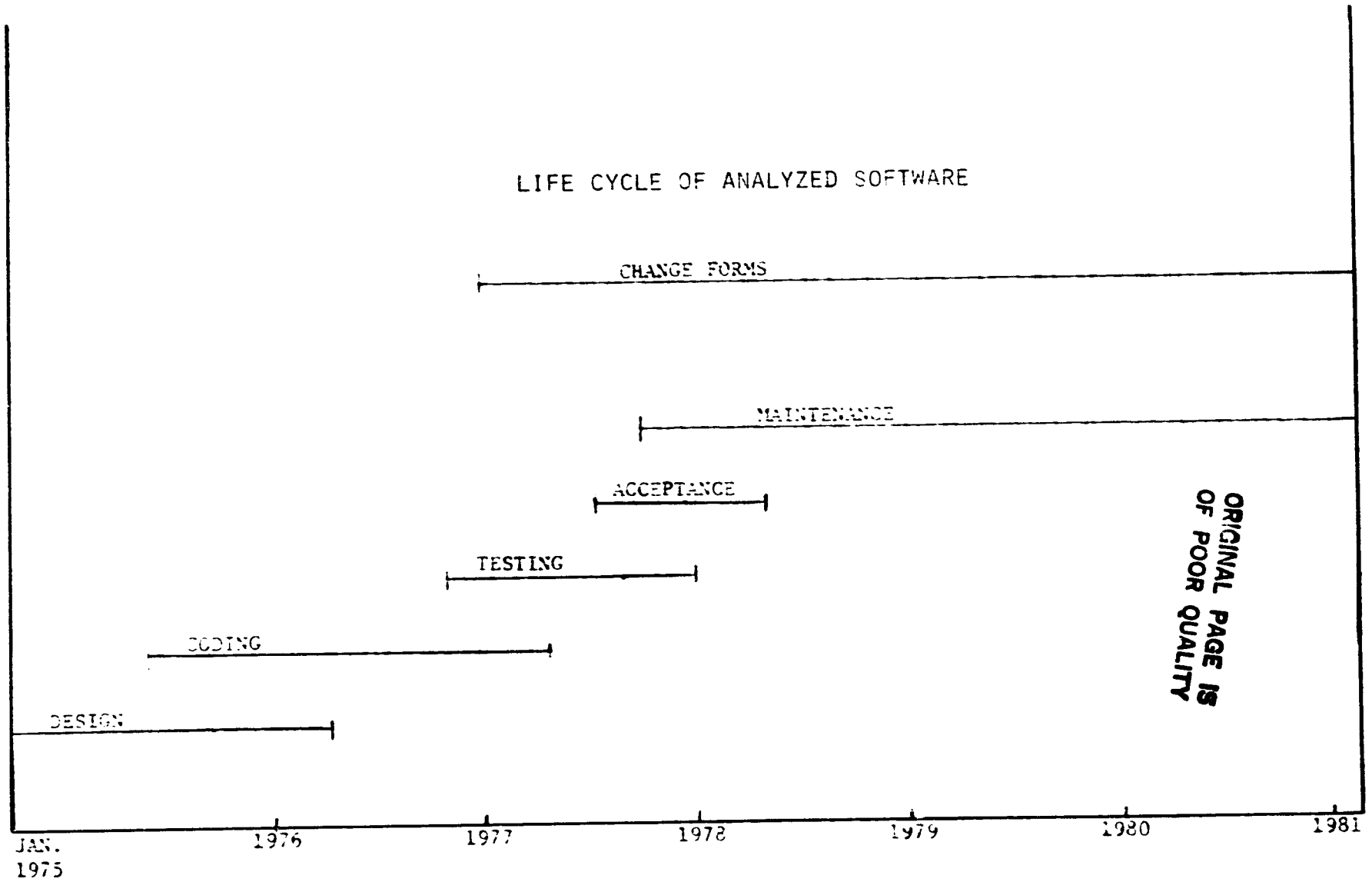
49% ERRORS IN MODIFIED MODULES

51% ERRORS IN NEW MODULES

ERROR CORRECTIONS VS. MODIFICATIONS

38% OF CHANGES WERE MODIFICATIONS

62% OF CHANGES WERE ERROR CORRECTIONS



NUMBER MODULES

NUMBER OF LINES	ALL MODULES		MODULES WITH ERRORS	
	SOURCE	EXECUTABLE	SOURCE	EXECUTABLE
0-50	53	258	3	49
51-100	107	70	16	25
101-150	80	26	20	13
151-200	56	13	19	7
201-250	34	1	12	1
251-300	14	1	9	0
301-350	7	1	4	1
351-400	9	0	7	0
>400	10	0	6	0
TOTAL	370	370	96	96

NUMBER OF MODULES AFFECTED BY AN ERROR (DATA SET: 211 TEXTUAL ERRORS
174 CONCEPTUAL ERRORS)

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

RESULTS: SIMILAR TO OTHER STUDIES, FEW ERRORS INVOLVE MORE THAN ONE MODULE

	NUMBER OF ERRORS (12 TOTAL)	AVERAGE EFFORT TO CORRECT
MISUNDERSTOOD OR INCORRECT REQUIREMENTS	8	32 HOURS
INCORRECT DESIGN OR IMPLEMENTATION OF A MODULE	3	0.5 HOURS
CLERICAL ERROR	1	0.5 HOURS

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

NUMBER OF ERRORS PER MODULE (DATA SET: 215 ERRORS)

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

C-2

	NUMBER OF ERRORS (15 TOTAL)	AVERAGE EFFORT TO CORRECT
MISUNDERSTOOD OR INCORRECT SPECIFICATIONS	8	24 HOURS
INCORRECT DESIGN OR IMPLEMENTATION OF A MODULE COMPONENT	5	16 HOURS
CLERICAL ERROR	2	4.5 HOURS

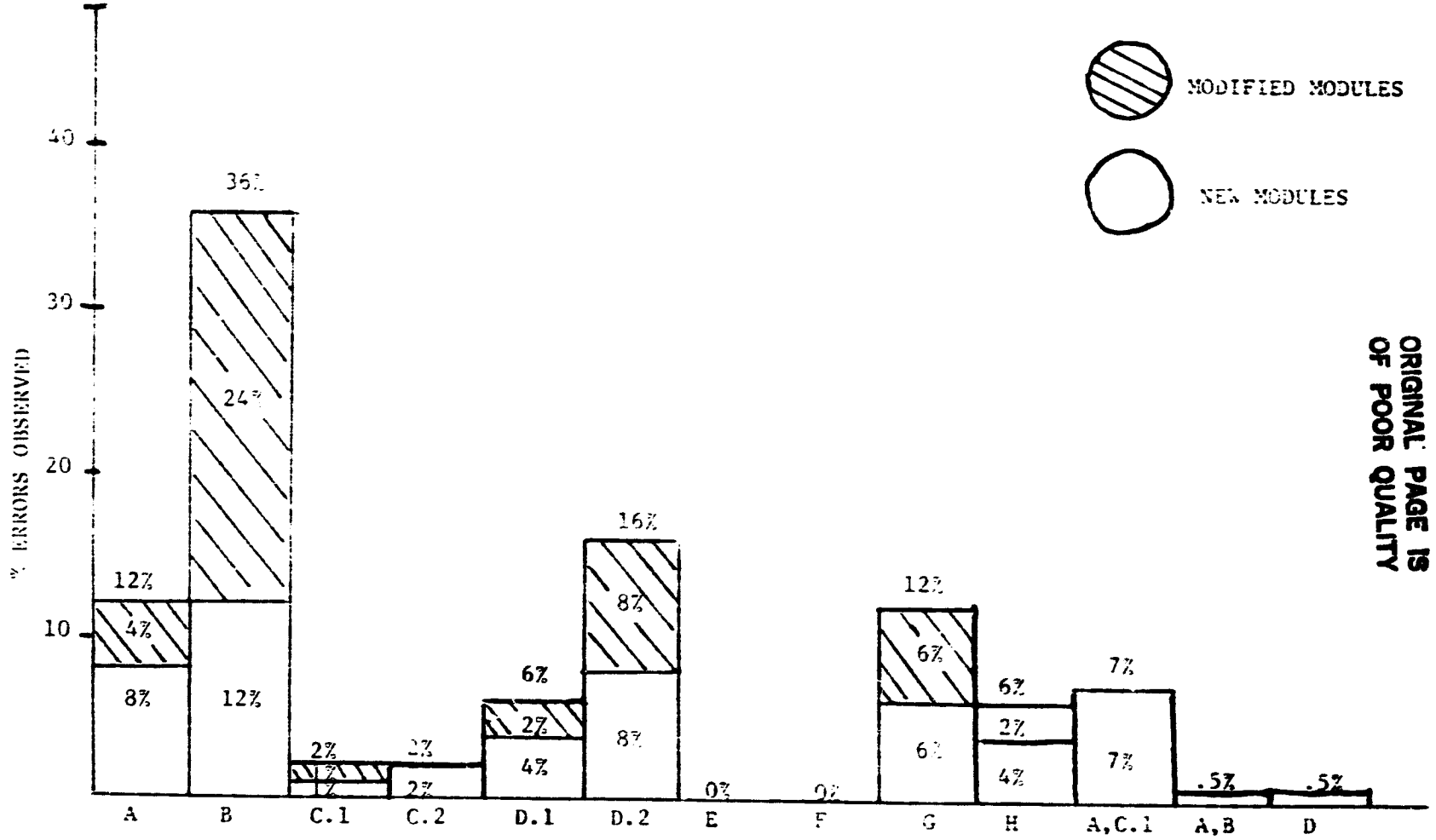
EFFORT TO CORRECT ERRORS IN THREE MOST ERROR PRONE MODIFIED MODULES

ERROR DISTRIBUTION BY TYPE

CATEGORIES:

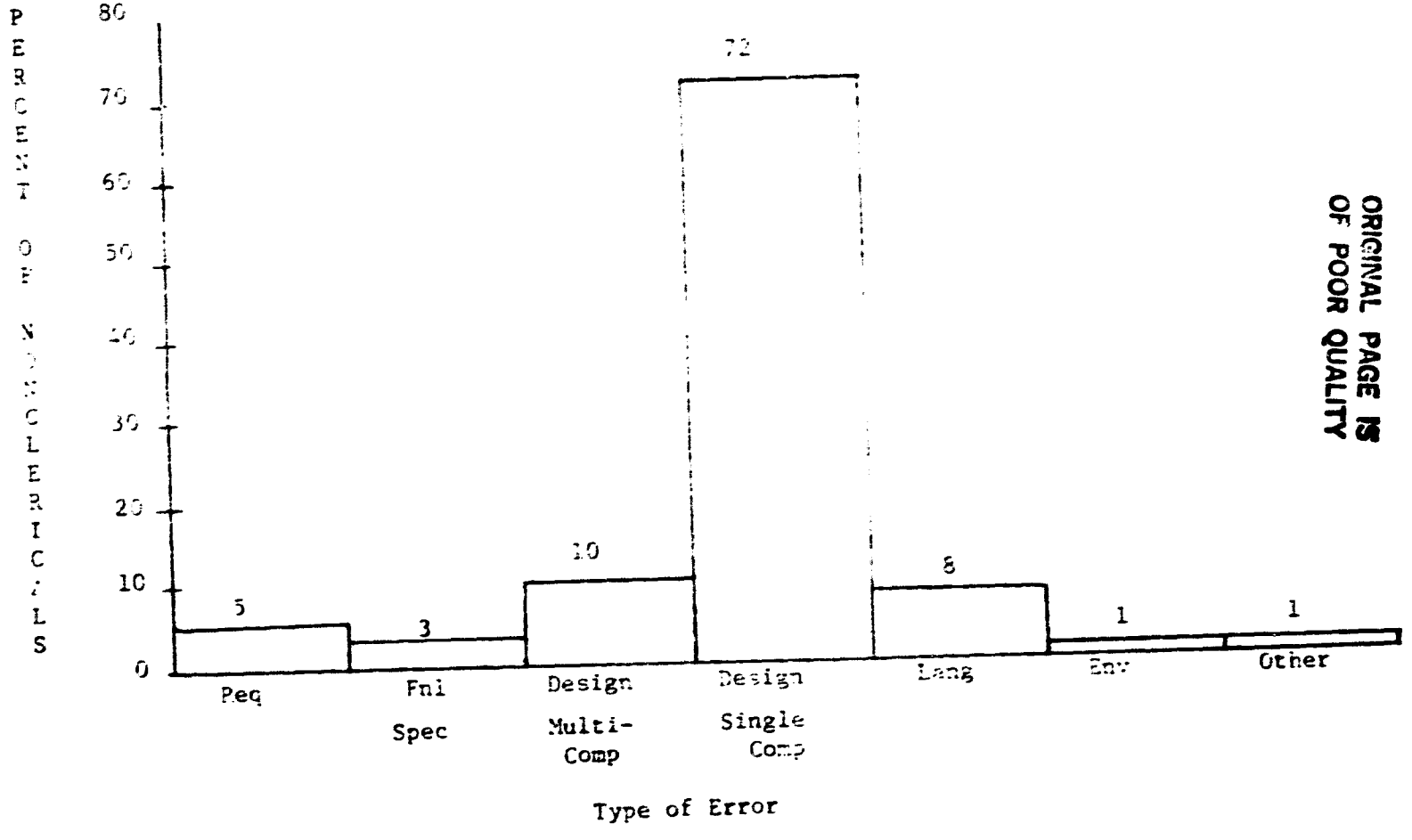
- A: REQUIREMENTS INCORRECT OR MISINTERPRETED
- B: FUNCTIONAL SPECIFICATION INCORRECT OR MISINTERPRETED
- C: DESIGN ERROR INVOLVING SEVERAL COMPONENTS
- D: DESIGN ERROR IN A SINGLE COMPONENT
- E: MISUNDERSTANDING OF EXTERNAL ENVIRONMENT
- F: ERRORS IN PROGRAMMING LANGUAGE OR COMPILER
- G: CLERICAL ERROR
- H: ERROR DUE TO PREVIOUS MISCORRECTION OF AN ERROR

SOURCES OF ERRORS



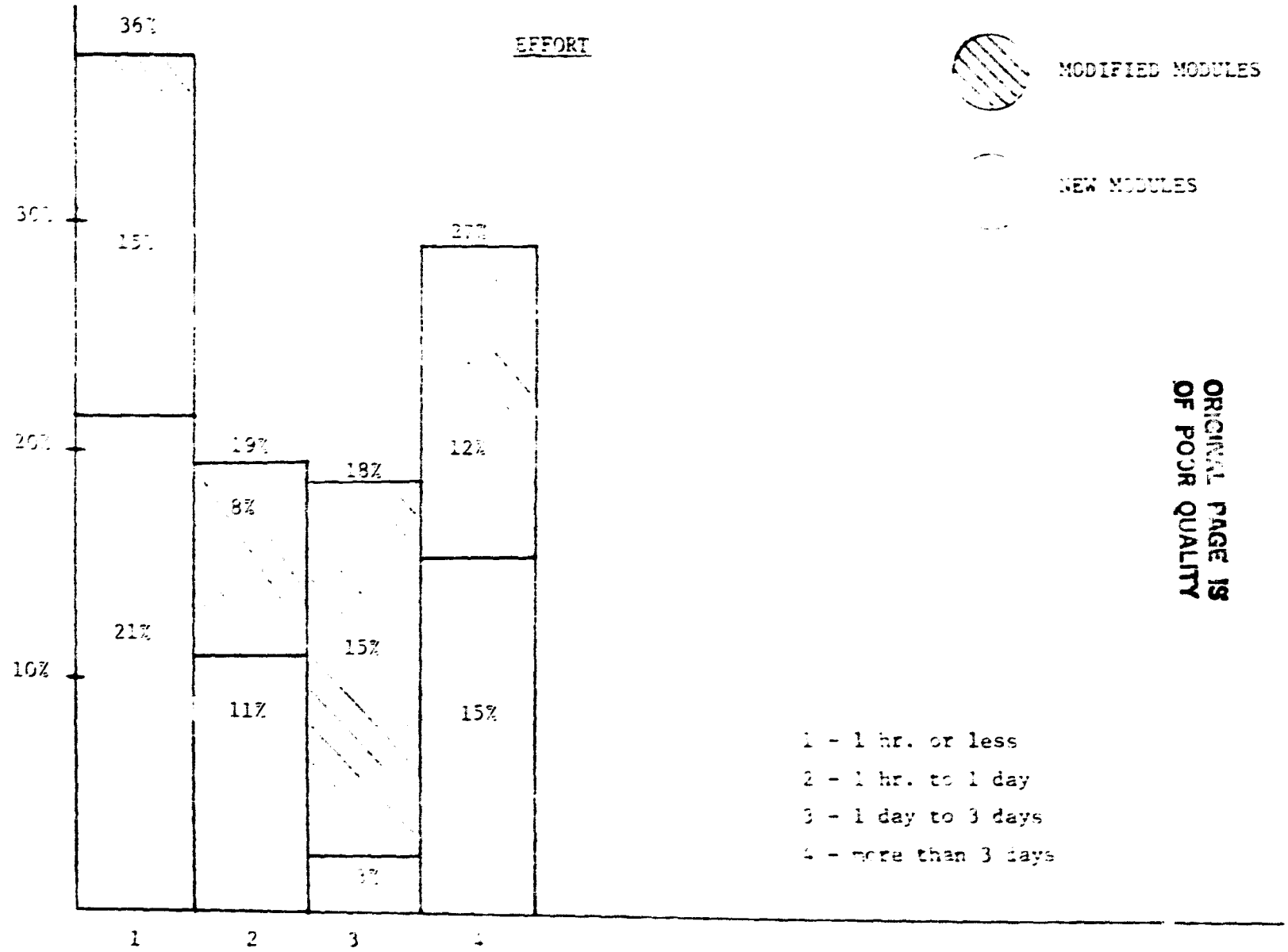
ORIGINAL PAGE IS
OF POOR QUALITY

RESULTS: SIMILAR TO ENDRES' STUDY (46% VS. 48% HERE INVOLVED MISUNDERSTANDING OF THE PROBLEM



SEL2 SOURCES OF NONCLERICAL ERRORS

From previous slide
adjusted for differences
in counting schemes



ABSTRACT ERROR TYPES

CATEGORIES:

- INITIALIZATION - FAILURE TO INITIALIZE DATA ON ENTRY/EXIT
- CONTROL STRUCTURE - INCORRECT PATH TAKEN
- INTERFACE - ASSOCIATED WITH STRUCTURES OUTSIDE MODULES ENVIRONMENT
- DATA - INCORRECT USE OF A DATA STRUCTURE
- COMPUTATION - ERRONEOUS EVALUATION OF A VARIABLE'S VALUE
- COMMISSION - INCORRECT EXECUTABLE STATEMENT
- OMISSION - NEGLECTING TO INCLUDE SOME ENTITY IN A MODULE

RESULT: LARGEST PERCENT OF ERRORS INVOLVE INTERFACE (39%)
CONTROL MORE OF A PROBLEM IN NEW MODULES
DATA AND INITIALIZATION MORE OF A PROBLEM IN MODIFIED MODULES
SMALL NUMBER OF OMISSION ERRORS IN MODIFIED MODULES

MIGHT IMPLY - BASIC ALGORITHMS FOR THE MODIFIED MODULES WERE CORRECT BUT NEEDED SOME ADJUSTMENT WITH RESPECT TO DATA VALUES AND INITIALIZATION FOR THE APPLICATION OF THE OLD ALGORITHM TO THE NEW APPLICATION

	COMMISSION		OMISSION	
	NEW	MODIFIED	NEW	MODIFIED
INITIALIZATION	2	9	5	9
CONTROL	12	2	16	6
INTERFACE	23	31	27	6
DATA	10	17	1	3
COMPUTATION	16	21	3	3
	28%	36%	23%	12%
	64%		35%	

	TOTAL		
	NEW	MODIFIED	
INITIALIZATION	7	18 --- 25	(11%)
CONTROL	28	8 --- 36	(16%)
INTERFACE	50	37 --- 87	(39%)
DATA	11	20 --- 31	(14%)
COMPUTATION	19	24 --- 43	(19%)
	115	107	

ABSTRACT CLASSIFICATION OF ERRORS

MODULE SIZE	ERRORS/1000 LINES
50	16.0
100	12.6
150	12.4
200	7.6
> 200	6.4

ERRORS/1000 EXECUTABLE LINES (INCLUDES ALL MODULES)

EXPLANATIONS:

INTERFACE ERRORS SPREAD ACROSS ALL MODULES

MAJORITY OF MODULES EXAMINED WERE SMALL BIASING THE RESULT

LARGER MODULES WERE CODED WITH MORE CARE

ERRORS IN SMALLER MODULES WERE MORE APPARENT

MODULE SIZE	AVERAGE CYCLOMATIC COMPLEXITY
50	6.0
100	17.9
150	28.1
200	52.7
> 200	60.0

AVERAGE CYCLOMATIC COMPLEXITY FOR ALL MODULES

MODULE SIZE	AVERAGE CYCLOMATIC COMPLEXITY	ERRORS/1000 EXECUTABLE LINES
50	6.2	65.0
100	19.6	33.3
150	27.5	24.6
200	56.7	13.4
>200	77.5	9.7

COMPLEXITY AND ERROR RATE FOR ERRORED MODULES

RESULT: AVERAGE CYCLOMATIC COMPLEXITY GREW FASTER THAN SIZE

CONCLUSIONS

ERROR ANALYSIS PROVIDES USEFUL INFORMATION

- CAN SEE NEW APPLICATION WITH CHANGING REQUIREMENTS
- INSIGHTS INTO DIFFERENT ERRORS FOR NEW AND MODIFIED MODULES
- MAJOR ERROR PROBLEMS WITH DIFFERENT APPLICATION EXPERIENCE
- CAN COMPARE ENVIRONMENTS

MODULE SIZE AN OPEN QUESTION WRT ERRORS

- THE LARGER THE MODULE (WITHIN LIMITS) THE LESS ERROR PRONE
- WE ARE NOT READY TO PUT ARTIFICIAL LIMITS

RECOMMENDATIONS:

- THE ENVIRONMENT MUST BE BETTER UNDERSTOOD
- MORE DATA MUST BE COLLECTED
- MORE STUDIES MADE

D3

N83 32359

WHEN AND HOW TO USE A SOFTWARE RELIABILITY MODEL

Amrit L. Goel[1], Victor R. Basili[2],
and Peter M. Valdes[3]

Many analytical models were proposed during the last decade for software reliability assessment. These models served a useful purpose in identifying the need for an objective approach to determining the quality of a software system as it goes through various stages of development. However, by and large, these models have not been as widely and convincingly used as was expected.

In this paper we attempt to identify the causes of this state of affairs and suggest some remedial actions. For example, we feel that very often the models are used without a clear understanding of their underlying assumptions and limitations. Also, there seems to be some misunderstanding about the interpretations of model inputs and outputs. To overcome some of these difficulties, we provide a classification of the available models and suggest which types of models are applicable in a given phase of the software development cycle.

The work reported in this paper represents the first step towards developing a general methodology for assessing software quality and reliability throughout the development cycle. Further work on this topic will be published in the near future.

---

[1]Professor of Industrial Engineering and Operations Research; and Computer and Information Science, Syracuse University. Visiting Professor, University of Maryland, College Park, MD.

[2]Chairman and Professor, Dept. of Computer Science, University of Maryland, College Park, MD.

[3]Graduate Assistant, University of Maryland.

THE VIEWGRAPH MATERIALS

for the

A. GOEL PRESENTATION FOLLOW

WHEN AND HOW TO USE A SOFTWARE RELIABILITY MODEL

AMRIT L. GOEL, VICTOR R. BASILI,
AND PETER M. VALDES

SEVENTH ANNUAL SOFTWARE ENGINEERING WORKSHOP
NASA/GSFC

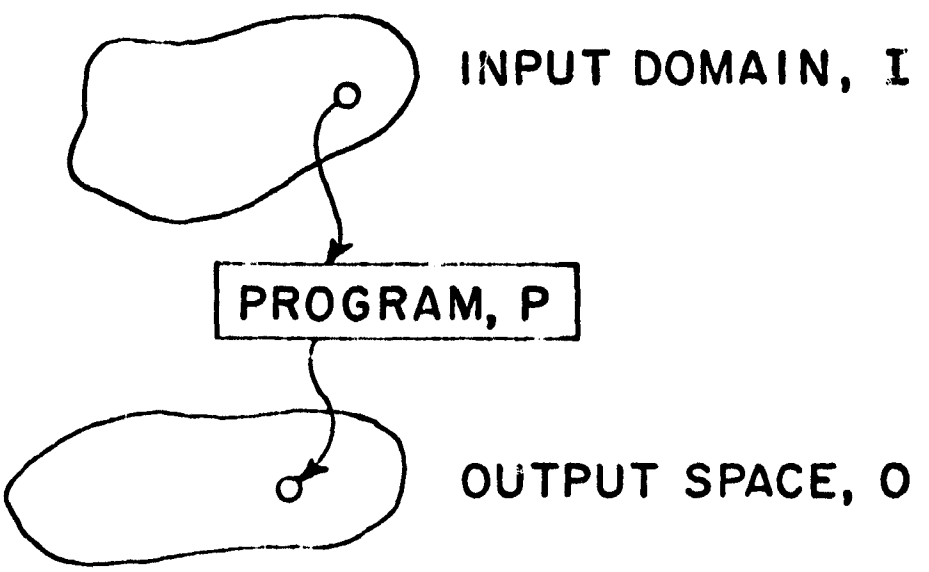
DECEMBER 1, 1982

OUTLINE

- SOFTWARE RELIABILITY
- SOFTWARE RELIABILITY MODELS
  - CLASSIFICATION
- SOFTWARE DEVELOPMENT PHASES
- APPLICABILITY OF MODELS IN EACH PHASE
- DISCUSSION OF MAJOR MODEL ASSUMPTIONS

SOFTWARE

SOFTWARE (ALSO CALLED PROGRAM) IS ESSENTIALLY AN INSTRUMENT FOR TRANSFORMING A DISCRETE SET OF INPUTS (FROM INPUT DOMAIN) INTO A DISCRETE SET OF OUTPUTS (INTO ITS OUTPUT SPACE)



SOFTWARE ERROR

SOFTWARE ERROR IS A DISCREPANCY BETWEEN WHAT THE SOFTWARE DOES AND WHAT THE USER OR THE COMPUTING ENVIRONMENT (PHYSICAL MACHINE, O/S, COMPILER, ETC.) WANTS IT TO DO.

SOFTWARE RELIABILITY

- THE PROBABILITY THAT SOFTWARE WILL NOT CAUSE THE FAILURE OF A SYSTEM TO PERFORM A REQUIRED TASK OR MISSION FOR A SPECIFIED TIME IN A SPECIFIED ENVIRONMENT.

- AN ATTRIBUTE OF SOFTWARE QUALITY PERTAINING TO THE EXTENT TO WHICH A COMPUTER PROGRAM CAN BE EXPECTED TO PERFORM ITS INTENDED FUNCTION WITH REQUIRED PRECISION.

SOFTWARE RELIABILITY

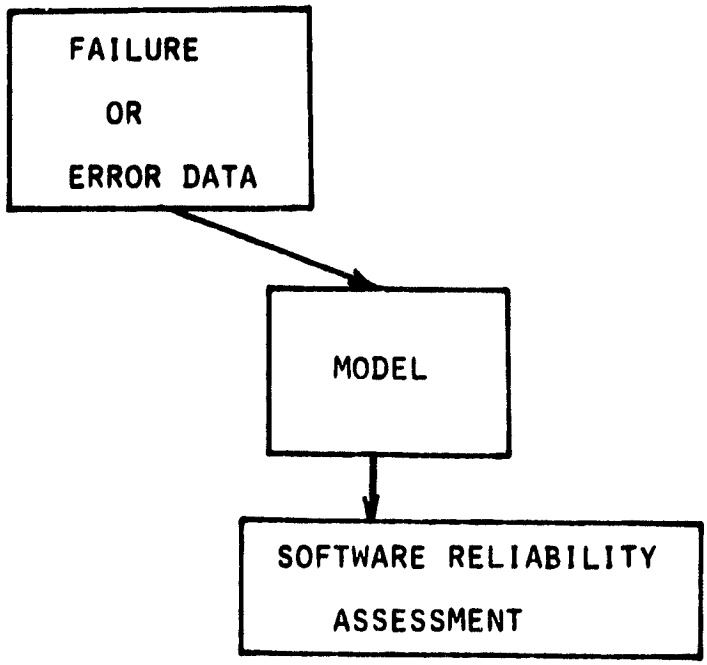
LET E BE A CLASS OF ERRORS OF INTEREST AND T BE A MEASURE OF RELEVANT TIME (UNITS DETERMINED BY THE APPLICATION AT HAND).

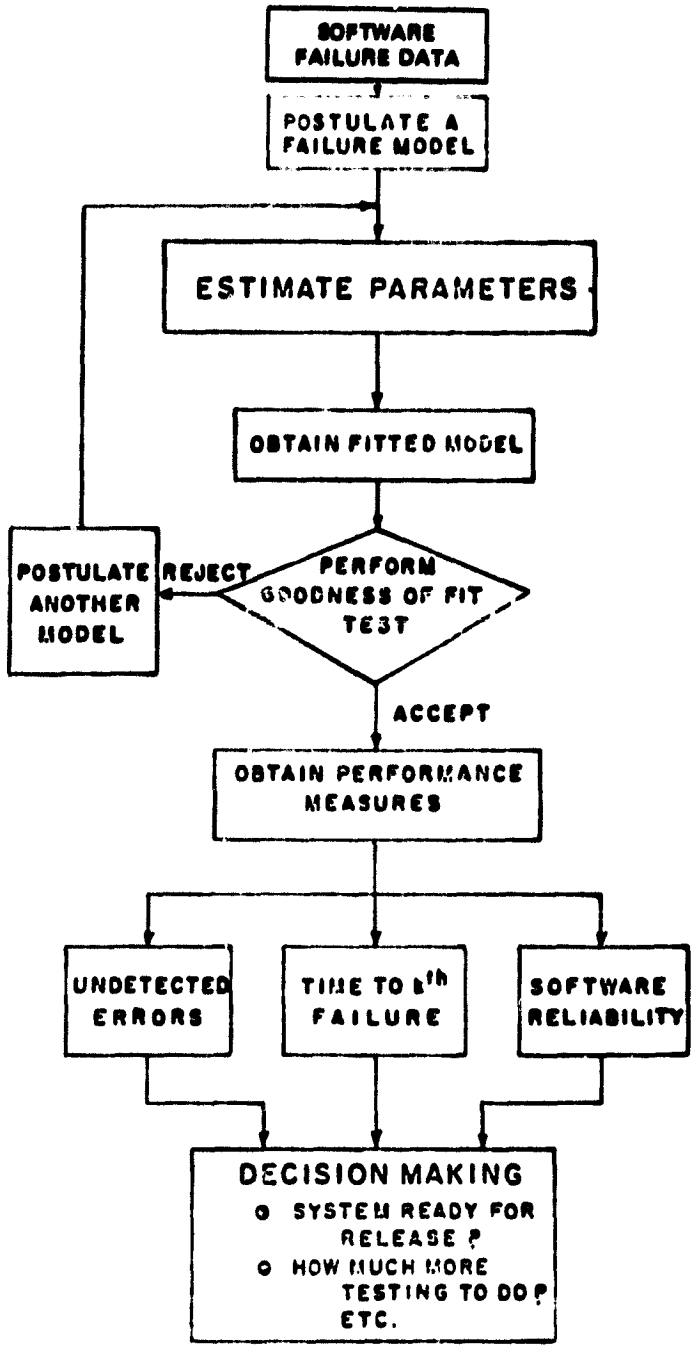
THEN THE RELIABILITY OF A SOFTWARE PACKAGE WITH RESPECT TO THE CLASS OF ERRORS E AND WITH RESPECT TO THE METRIC T IS THE PROBABILITY THAT NO ERROR OF THE CLASS OCCURS DURING THE EXECUTION OF THE PROGRAM FOR A PRESPECIFIED PERIOD OF RELEVANT TIME.

NEED FOR SOFTWARE RELIABILITY, ASSESSMENT

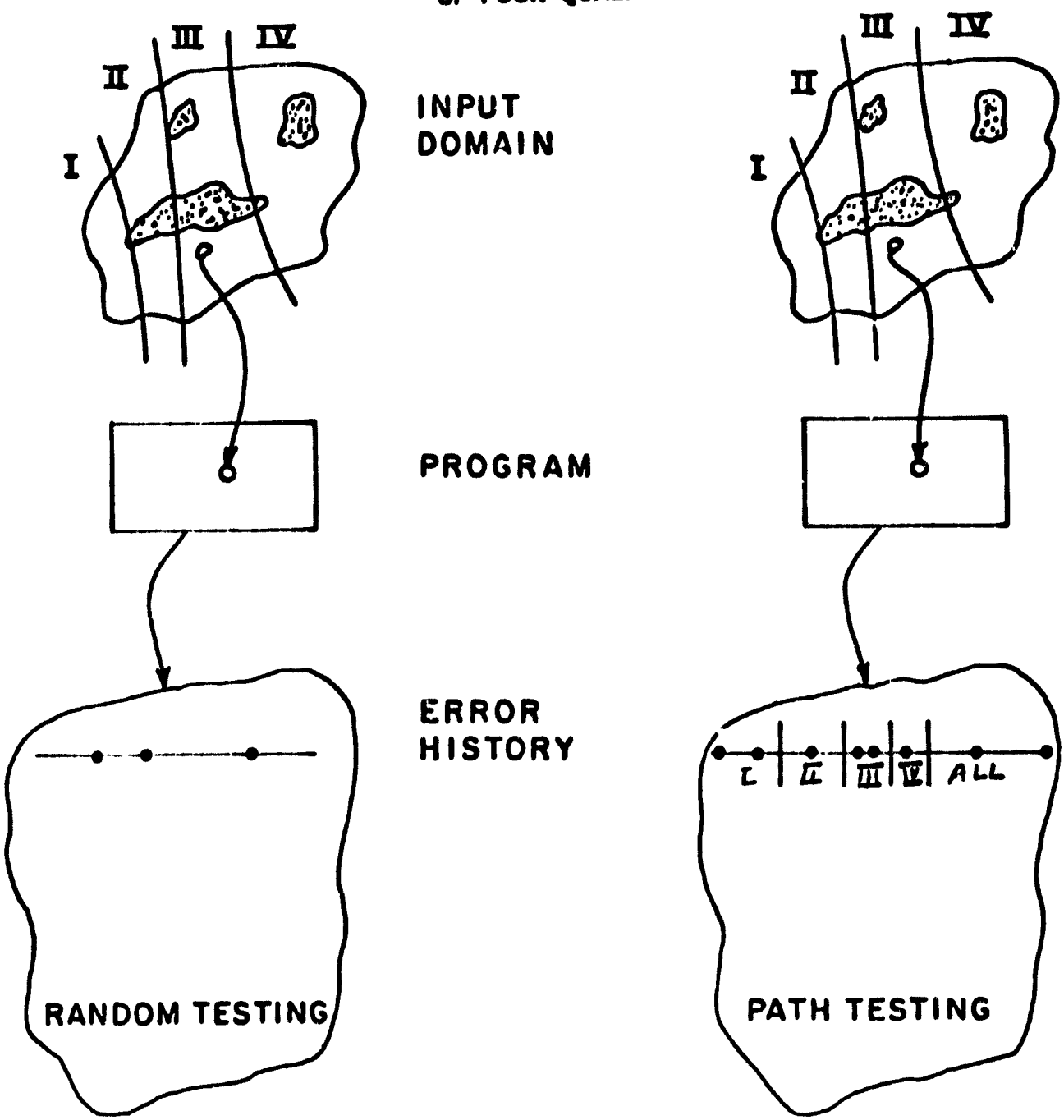
- ESTIMATE POTENTIAL RELIABILITY DURING CONCEPTUAL PHASE
- ESTABLISH REALISTIC NUMERICAL RELIABILITY GOALS DURING DEFINITION PHASE
- ESTABLISH EXISTING LEVELS OF ACHIEVED RELIABILITY
- MONITOR PROGRESS TOWARD ACHIEVING SPECIFIED RELIABILITY GOALS OR REQUIREMENTS
- ESTABLISH RELIABILITY CRITERIA FOR FORMAL QUALIFICATION

GENERAL APPROACH





FLOWCHART FOR SOFTWARE FAILURE DATA ANALYSIS AND DECISION MAKING



TESTING PROCESS AND ERROR HISTORY

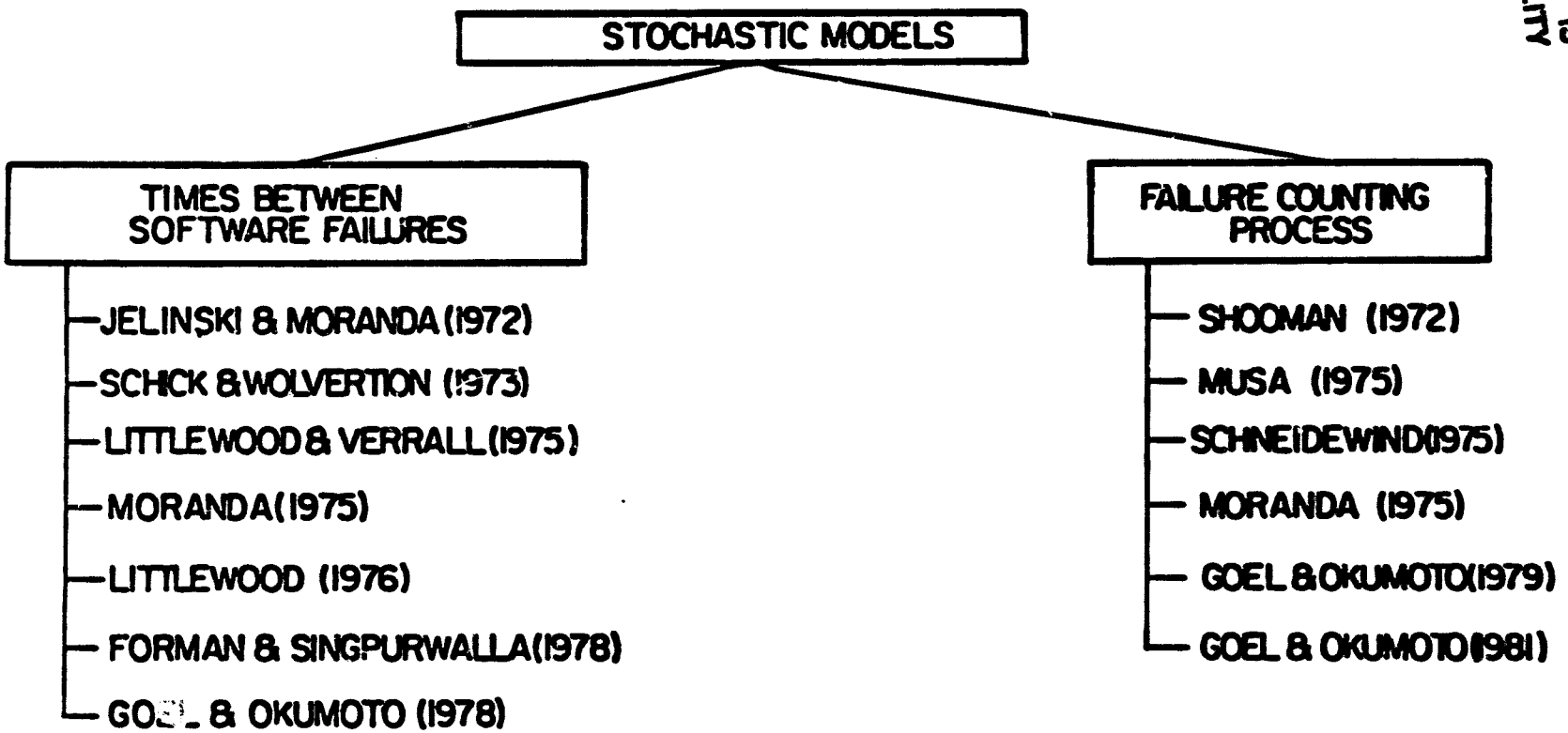
SOFTWARE RELIABILITY MODELS

TIME-DEPENDENT MODELS

ASSUMPTIONS OF MODELS EMPHASIZING DETECTION PROCESS

- FAILURES ARE INDEPENDENT
- NUMBER OF FAILURES IS CONSTANT
- EACH FAILURE IS REPAIRED BEFORE TESTING CONTINUES
- INPUTS WHICH EXERCISE THE PROGRAM ARE RANDOMLY SELECTED
- ALL FAILURES ARE OBSERVABLE
- TESTING IS OF UNIFORM INTENSITY AND REPRESENTATIVE OF OPERATIONAL ENVIRONMENT
- FAILURE RATE AT ANYTIME IS PROPORTIONAL TO CURRENT NUMBER OF FAILURES

OVERVIEW OF SOFTWARE RELIABILITY MODELS



SOFTWARE RELIABILITY MODELS

TIME INDEPENDENT MODELS

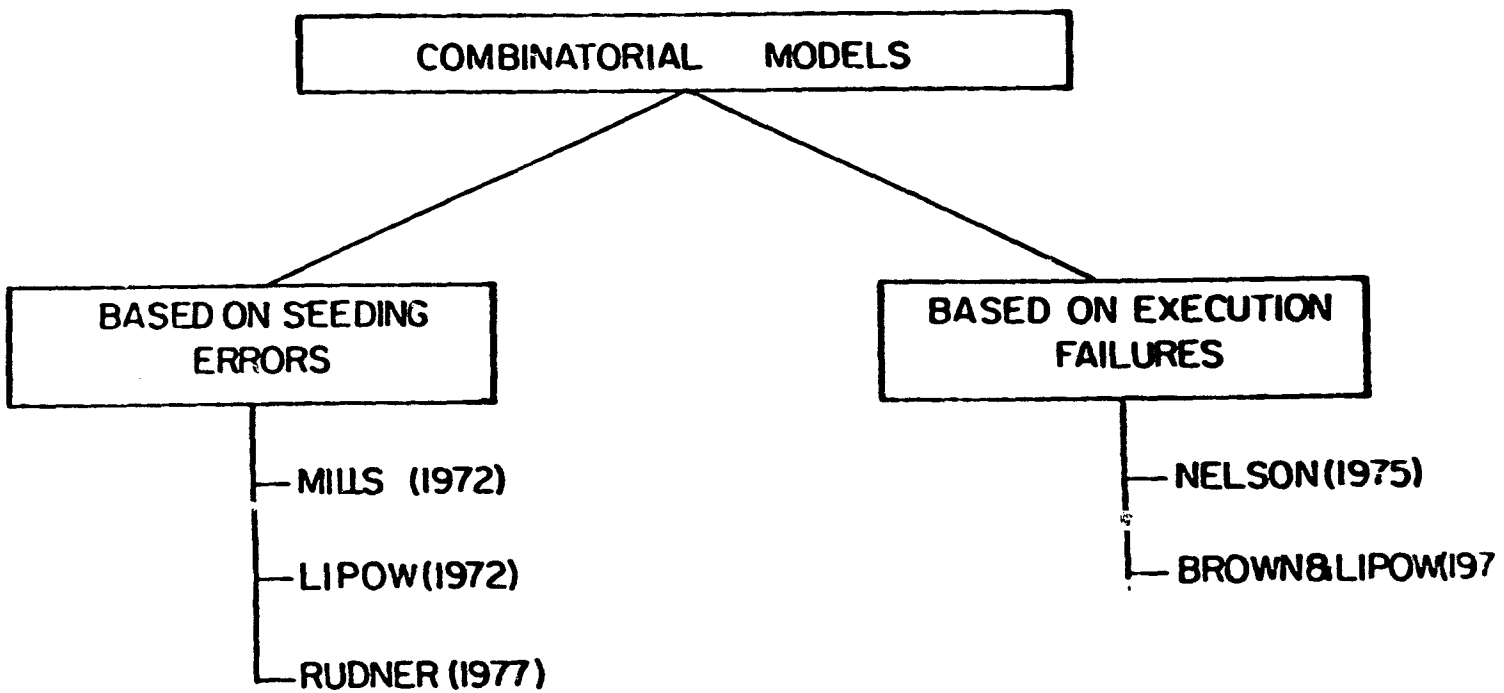
- USE OBSERVED RESULTS OF EXPERIMENTS CONDUCTED ON ELEMENTS OF THE PROGRAM'S INPUT SPACE

- USE A-PRIORI KNOWLEDGE OF INPUT SPACE

- TWO CLASSES

  ERROR SEEDING

  INPUT SPACE SAMPLING


COMBINATORIAL MODELS
BASED ON SEEDING ERRORS
BASED ON EXECUTION FAILURES
MILLS (1972)
LIPOW(1972)
RUDNER (1977)
NELSON(1975)
BROWN&LIPOW(197

ASSUMPTIONS

I. TIMES BETWEEN FAILURE MODELS

- INDEPENDENT INTERFAILURE TIMES
- EQUAL PROBABILITY OF EXPOSING EMBEDDED ERRORS
- ERRORS EMBEDDED ARE INDEPENDENT
- TIME-DEPENDENCE
- IMMEDIATE ERROR REMOVAL, PERFECT ERROR REMOVAL, NONINTRODUCTION OF NEW ERRORS
- RELIABILITY BASED ON REMAINING NUMBER OF ERRORS

II. FAILURE COUNTING MODELS

- ERRORS IN NONOVERLAPPING TIME INTERVALS ARE INDEPENDENT
- FAILURE RATE PROPORTIONAL TO EXPECTED ERROR CONTENT
- DECREASING FAILURE RATE WITH TIME (DISCRETE OR CONTINUOUS)

III. ERROR-SEEDING MODELS

- INDIGENOUS AND SEEDED ERRORS HAVE EQUAL PROBABILITY OF BEING DETECTED

IV. INPUT DOMAIN BASED MODELS

- INPUT PROFILE DISTRIBUTION IS KNOWN
- RANDOM TESTING IS USED
- INPUT DOMAIN CAN BE PARTITIONED INTO EQUIVALENCE CLASSES

SOME LIMITATIONS OF MOST MODELS

- INDEPENDENCE OF TIMES BETWEEN FAILURES
- EQUAL IMPORTANCE TO DIFFERENT TYPES OF ERRORS
- SAME FAILURE RATE FOR EACH ERROR
- NO PROVISION FOR INTRODUCTION OF NEW ERRORS
- DECREASING FAILURE RATE DURING DEBUGGING OR OPERATION

INDEPENDENT INTERFAILURE TIMES

NOT A REALISTIC ASSUMPTION IN GENERAL, ESPECIALLY WHEN THE TESTING PROCESS IS NOT RANDOM. TIME TO NEXT FAILURE MAY VERY WELL DEPEND ON THE NATURE OF THE PREVIOUS FAILURE. IF THE PREVIOUS ERROR WAS CRITICAL, WE MIGHT INTENSIFY TESTING AND LOOK FOR ADDITIONAL CRITICAL ERRORS, WHICH IMPLIES NON-INDEPENDENT INTERFAILURE TIMES.

NHPP TYPE MODELS ARE ROBUST TO SUCH LACK OF INDEPENDENCE.

SOFTWARE FAILURE RATE IS PROPORTIONAL TO NUMBER OF REMAINING ERRORS

DOES NOT HOLD IN MANY CASES.

REMAINING ERRORS THAT RESIDE IN THE FREQUENTLY USED PORTION OF THE CODE ARE MORE LIKELY TO BE DETECTED THAN OTHERS.

IF, HOWEVER, TESTING IS REPRESENTATIVE OF USE, FAILURE RATE COULD BE CONSIDERED PROPORTIONAL TO ERROR CONTENT.

ERRORS DETECTED ARE IMMEDIATELY CORRECTED

NOT A REALISTIC ASSUMPTION IN MOST PRACTICAL SITUATIONS.

CORRECTION PROCESS DOES NOT INTRODUCE NEW ERRORS

VERY RARELY SATISFIED IN PRACTICE. A PARTIAL SOLUTION WAS ATTEMPTED IN THE IMPERFECT DEBUGGING MODEL, BUT A GENERAL SOLUTION IS NOT AVAILABLE.

TESTING PROCESS IS REPRESENTATIVE OF OPERATIONAL ENVIRONMENT

THIS IS RARELY TRUE. WE PREFER A RELIABILITY MEASURE BASED ON USER REQUIREMENTS RATHER THAN A SIMPLE UNCONDITIONED SOFTWARE RELIABILITY MEASURE.

USE OF EXECUTION TIME BETWEEN FAILURES

HAVE TO USE IT WITH CAUTION. ONE DEBUGGER COULD RUN AND RERUN THE PROGRAM TO UNCOVER REMAINING ERRORS CAUSING HIGH EXECUTION TIME BETWEEN FAILURES WHILE ANOTHER ONE MIGHT ANALYZE THE PROGRAM IN DETAIL AND THEN RUN THE (SAME) PROGRAM JUDICIOUSLY. FORMER CASE WOULD GIVE A WRONG IMPRESSION OF HIGHER RELIABILITY.

INCREASING FAILURE RATE BETWEEN FAILURES

CONTRARY TO THE ASSUMPTION THAT SOFTWARE DOES NOT WEAR OUT. BUT, THIS WOULD BE SO IF TESTING INTENSITY INCREASES DURING SUCH INTERVALS. OVERALL, NOT A REALISTIC ASSUMPTION.

SOFTWARE DEVELOPMENT PHASES

DESIGN

UNIT TESTING

INTEGRATION TESTING

ACCEPTANCE TESTING

OPERATION

APPLICABILITY OF EXISTING SOFTWARE RELIABILITY MODELS

I. DESIGN

- EXISTING MODELS NOT APPLICABLE

II. UNIT TESTING

- SEEDING MODELS APPLICABLE IF WE CAN ASSUME THAT INDIGENOUS AND SEEDED ERRORS HAVE EQUAL PROBABILITIES OF DETECTION.
- INPUT DOMAIN BASED MODELS MAY BE APPLICABLE.
- TBF AND FC MODELS NOT APPLICABLE.

III. INTEGRATION TESTING

- ALL MODELS APPLICABLE IF RANDOM TESTING IS USED.
- FC MODELS MAY BE APPLICABLE FOR DETERMINISTIC TESTING.

IV. ACCEPTANCE TESTING

- INPUT DOMAIN BASED MODELS APPLICABLE.
- ERROR SEEDING MODELS NOT APPLICABLE
- TBF AND FC MODELS DO NOT SEEM TO BE APPLICABLE AS ERRORS ARE NOT IMMEDIATELY CORRECTED; SOME TBF AND FC MODELS MAY BE ROBUST TO THIS REQUIREMENT

V. OPERATION

- INPUT DOMAIN MODELS MAY BE APPLICABLE PROVIDED USER INPUTS ARE RANDOM FROM THE INPUT PROFILE DISTRIBUTION.

DESIGN PHASE

- USER REQUIREMENTS ARE TRANSFORMED TO COMPUTER COMPATIBLE SPECIFICATIONS.
- DESIGN ERRORS MAY BE CORRECTED BY VISUAL INSPECTION OR BY OTHER INFORMAL PROCEDURES.
- EXISTING SOFTWARE RELIABILITY MODELS ARE NOT APPLICABLE AT THIS STAGE BECAUSE
  - TEST CASES TO EXPOSE ERRORS REQUIRED BY SEEDING AND INPUT DOMAIN BASED MODELS DO NOT EXIST
  - ERROR HISTORY REQUIRED BY TIMES BETWEEN FAILURES AND FAILURE COUNT MODELS DOES NOT EXIST

UNIT TESTING

EACH MODULE HAS ITS OWN SPECIFIED INPUT DOMAIN AND OUTPUT SPECIFICATION.

MODULE SPECIFICATION IS TRANSFORMED INTO A PROGRAM (CODING).

TEST CASES BASED ON THE INPUT DOMAIN AND OUTPUT SPECIFICATION ARE DESIGNED TO EXPOSE ERRORS. THE TEST CASES DO NOT USUALLY FORM A REPRESENTATIVE SAMPLE OF THE OPERATIONAL PROFILE DISTRIBUTION.

TIMES BETWEEN EXPOSURE OF ERRORS ARE NOT RANDOM SINCE TEST CASES ARE EXECUTED AND DESIGNED IN A DETERMINISTIC FASHION.

EXPOSED ERRORS ARE CORRECTED (DEBUGGED).

UNIT TESTING: RELIABILITY MODELS

SEEDING MODELS ARE APPLICABLE IF WE CAN ASSUME THAT INDIGENOUS AND SEEDED ERRORS HAVE EQUAL PROBABILITIES OF DETECTION

INPUT DOMAIN BASED MODELS MAY BE APPLICABLE

IF TESTS CAN BE MATCHED WITH THE OPERATIONAL PROFILE DISTRIBUTION

TBF AND FC MODELS NOT APPLICABLE

INTEGRATION TESTING

MODULES ARE INTEGRATED INTO SUBSYSTEMS OR INTO THE WHOLE SYSTEM.

TEST CASES ARE GENERATED TO VERIFY THE CORRECTNESS OF THE WHOLE SYSTEM.

DUE TO THE COMPLEXITY OF THE INTEGRATED SYSTEM, TEST CASES MAY BE GENERATED

- RANDOMLY (BASED ON AN INPUT PROFILE DISTRIBUTION);
- DETERMINISTICALLY (BASED ON A SET OF TEST CRITERIA).

EXPOSED ERRORS ARE CORRECTED. HOWEVER, ADDITIONAL ERRORS MAY BE INTRODUCED.

INTEGRATION TESTING: RELIABILITY MODELS

ALL MODELS APPLICABLE IF RANDOM TESTING IS USED.

FAILURE COUNT MODELS MAY BE ROBUST TO LACK OF INDEPENDENCE AND COULD BE USED FOR DETERMINISTIC TESTING.

ACCEPTANCE TESTING

SOFTWARE IS GIVEN TO "FRIENDLY USERS."

THESE USERS GENERATE TEST CASES (USUALLY RANDOM) TO VERIFY SOFTWARE CORRECTNESS. THE GENERATED TEST CASES MAY BE ASSUMED REPRESENTATIVE OF THE OPERATIONAL PROFILE DISTRIBUTION.

USUALLY EXPOSED ERRORS ARE NOT IMMEDIATELY CORRECTED.

OPERATIONAL PHASE

SOFTWARE IS PUT INTO USE.

INPUTS MAY NOT BE RANDOM ANYMORE SINCE A USER MAY BE USING THE SAME SOFTWARE FUNCTION ON A ROUTINE BASIS. INPUT MAY BE CORRELATED.

ERRORS ARE NOT IMMEDIATELY CORRECTED. APPLICABLE MODELS (MAY NOT SATISFY ALL ASSUMPTIONS).

INPUT DOMAIN BASED MODELS.

PROBLEMS WITH RELIABILITY ASSESSMENT

SOMETIMES MODELS ARE USED (SUCCESSFULLY OR OTHERWISE) WITH INCOMPLETE UNDERSTANDING OF UNDERLYING ASSUMPTIONS AND LIMITATIONS.

ROBUSTNESS TO DEVIATIONS FROM ASSUMPTIONS IS NOT FULLY KNOWN.

APPLICABILITY OF MODELS IN DIFFERENT ENVIRONMENTS NEEDS FURTHER WORK.

MEASUREMENT (FOR RELIABILITY ASSESSMENT) IS DONE TOO LATE IN THE LIFE CYCLE.

NEED FOR MODEL SIMPLICITY (USABILITY) VS. CAPTURING DETAILS OF REALITY NOT FULLY APPRECIATED.

CURRENT ACTIVITIES

- EXAMINING RELIABILITY MEASURES ACROSS ALL LIFE CYCLE PHASES
- STUDYING EFFECTS OF TESTING ON RELIABILITY
- EXPLORING USE OF TEST CRITERIA AS MEASURES OF QUALITY AND RELIABILITY
- DEVELOPING RELATIONSHIPS BETWEEN DESIGN, COMPLEXITY, TESTING AND RELIABILITY

BASICALLY STUDYING THE ENTIRE LIFE CYCLE RATHER THAN JUST THE FINAL TESTING PHASE FOR QUALITY AND RELIABILITY ASSESSMENT.

D4

SOFTWARE PROTOTYPING IN THE SOFTWARE ENGINEERING LABORATORY

MARVIN V. ZELKOWITZ
DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF MARYLAND
COLLEGE PARK, MARYLAND 20742

INTRODUCTION

Over the last few years, several techniques have become popular withing the software engineering world. Concepts like "structured programming," "distributed processing," "expert systems," and others have all been proposed as a means to enhance software productivity. Recently the term "prototyping" has been applied to productivity improving (SEN80, SEN82). The NASA Goddard Software Engineering Laboratory is starting a project to evaluate prototyping within the NASA environment.

First of all, there are several definitions of a prototype. The dictionary defines it as an original or model on which something is based or formed. However, in looking at several computer glossaries through the year 1981, not one of them mentions a prototype software development. Thus the term is quite new and has yet to be standardized.

Prototyping is not modeling - another well used concept. In a model we are looking at only a few characteristics of an object. For example, in a wind tunnel, we are interested in the airstream past an airplane, not in its internal design. However, in a software prototype, we usually mean a complete working system, although it may be missing some functionality. Thus we are doing more than modeling, or its companion operation - simulation. We wish to build a system that demonstrates most of the behavior of the final product.

PROTOTYPING

In developing a prototype for NASA we need to understand what a prototype is. More importantly, for NASA, the issue of prototyping must answer the following questions:

What are the goals of a prototype? Is it to develop the requirements for a product? Evaluate its performance? Predict its final costs?

What are the issues involved? How does one design for a prototype? Does the software lifecycle change? Do we want multiple prototypes for different phases of the life cycle? How do we use a prototype when built?

What tools can be used to design a prototype? to build a prototype? to evaluate a prototype?

How does one measure a prototype? How do you know if your prototype was successful? Should you invest the cost and build the full system or abandon the project? What SHOULD a prototype cost? 10% of the final product or 50% or

even 100%?

The final question is does prototyping even fit into the NASA environment? Every software development environment is unique, and techniques which work in one environment might not work in another, so is talking about prototyping at NASA even relevant?

These are all questions which must be addressed, and the current project is one data point in evaluating its effectiveness.

RESEARCH ISSUES

WHAT IS A PROTOTYPE? There are several different models. In one it is a quick, dirty throw away implementation for evaluation purposes. The goals are to get something working quickly. This is often useful when the full requirements are not know well at the start and the prototype can be used to refine these requirements.

WHAT PROGRAMMING LANGUAGE SHOULD BE USED? There are several views as to the language that is to be used in a prototype. A low level language (e.g., Fortran, PL/I) can be used as the same implementation language for the full system. This leads to greater efficiency in the final prototype, but forces the programmer to design more details into the initial implementation.

There are several high level languages that have been proposed for prototyping. Snobol4 and SETL are two such examples. Both allow the programmer to avoid many details at a cost in execution speed. Unfortunately, these high level languages are not universally available and can not be used on all projects.

There is also research on very high level languages - often called specification or non-procedural languages. These specify what is to happen and not how, thus are good for a prototype where performance is not critical. However, these are still very experimental and not yet available in a production environment.

WHAT ARE PROGRAMMER CAPABILITIES? One unfortunate issue in the current studying of prototyping, is that it is a research topic being investigated by expert "supercoders". Once prototyped, a system is then built by "mere mortals". What will happen if prototyping becomes "an accepted" technique and mere mortals must build the initial design?

SOFTWARE ENGINEERING LABORATORY

So far the issue of prototyping has been described in very general terms. However, how does it apply to the NASA Software Engineering Laboratory? Within the Laboratory, three characteristics of software are under study: Profiles, Models and Methodologies. The effects of prototyping on each of these will be described.

PROFILES. One important aspect of the SEL is simply to measure software. Very little is generally know of a quantitative nature about software. This is certainly true of prototyping. One important goal is to simply add prototyping projects to the SEL data base in order to apply previous SEL analyses to this project as had been done to previous projects. Do cost models work? reliability models? error models? We need to simply characterize this software (SEL82).

MODELS. Once data is collected on prototyping projects, we need to evaluate models to see if they apply. Previously the SEL evaluated various cost models (Rayleigh, etc.). Do these apply to a prototype? Should they even apply? Is another model more appropriate?

METHODOLOGIES. Finally we need to revise the standard life cycle to account for prototypes. How are they designed, built and evaluated?

FLIGHT DYNAMICS ANALYSIS SYSTEM (FDAS)

At NASA a new product is being designed which seems like a good candidate for prototyping. This system, the Flight Dynamics Analysis System (FDAS), is being built to help experimenters try alternative flight dynamics models.

For example, today if an experiment is to be run (e.g., try a new orbit calculation model), the experimenter must access the Fortran source library, know which module to modify and make the changes, test the changes, recreate a new load module, and then run the experiment. The experimenter must have detailed knowledge of the software and the changes are a time consuming operation.

With FDAS, the experimenter enters the system, and an interactive dialogue, controlled by a data base, directs the experimenter to the correct module and aids in the change. Thus changes to software are easier, require less time and less expertise about the internals of the system.

Now why is this a good candidate for prototyping? In the past, software has generally been built for ground support software. Similar projects have been built for the last 15 to 20 years, thus NASA is an expert at such software. Issues like:

Requirements
Size
Execution characteristics
User interface
Algorithm design
Cost

are all well known (or as well known as is possible). Thus prototyping would not aid significantly. One can view all previous developments as "prototypes" for the next one.

However, FDAS is a very different system. Most of the factors mentioned above are unknown, so a prototype should aid greatly in this evaluation. In this case, the prototype has two functions: Refine the requirements so that a full FDAS implementation can be easily built, and test some of the design

ideas for feasibility.

In order to build the prototype, the following general strategy will be used:

(1) A subset of the requirements for FDAS will be written.

(2) A prototype will he built to these requirements.

(3) The prototype will be instrumented to collect usage and performance data.

(4) S.E.L. project data will be collected.

(5) The prototype will be evaluated.

(6) Features that are not effective will be redesigned.

(7) The full FDAS system will be built.

(8) The effectiveness of the prototype on the final product will be evaluated. Was FDAS cheaper to build? Will it be more reliable? Will it be more efficient? Will it have a better man/machine interface?

This evaluation will be by automated probes into the system. A logging file is being created for each user command. Execution characteristics will be added to this file as the prototype executes. A feature in the prototype to allow the user full range of changes to the software will be measured to see how often the experimenter must go "outside" of the commands provided by FDAS. This should greatly help in the user interface.

It is still too early in the development cycle of FDAS to give any conclusions. However, the project is moving along and a prototype should be ready for evaluation sometime midway into 1983. This should prove useful in addng to our knowledge about this important concept.

ACKNOWLEDGEMENT

This paper was supported by NASA grant NSG-5123 to the University of Maryland.

REFERENCES

(SEL82) Software Engineering Laboratory, Collected Papers - Volume 1, 1982.

(SEN80) ACM SIGSOFT Software Engineering Notes, Rancho Sante Fe Workshop, October, 1980

(SEN82) ACM SIGSOFT Software Engineering Notes, 2nd Software Engineering Symposium: Rapid Prototyping, December, 19821

THE VIEWGRAPH MATERIALS

for the

M. ZELKOWITZ PRESENTATION FOLLOW

JARGON

STRUCTURED PROGRAMMING

SOFTWARE ENGINEERING

DISTRIBUTED PROCESSING

DATA BASE

PROTOTYPING

PROTOTYPE

- THE ORIGINAL OR MODEL ON WHICH SOMETHING IS BASED OR FORMED

- SOMEONE OR SOMETHING THAT SERVES AS AN EXAMPLE OF ITS KIND

IN LOOKING AT SEVERAL COMPUTER GLOSSARIES UP THROUGH 1981, NO MENTION IS MADE OF PROTOTYPE.

USED IN:

1979 RANCHO SANTE FE WORKSHOP

1982 ACM SIGSOFT RAPID PROTOTYPING WORKSHOP

RECENT DOD REPORTS

SEVERAL THESES STARTING TO APPEAR ON TOPIC

A PROTOTYPE IS NOT A MODEL

- A MODEL USUALLY INVOLVES LOOKING AT ONLY A FEW CHARACTERISTICS
- A SIMULATION IS USUALLY A MODEL AND NOT A PROTOTYPE
- THE PROTOTYPE NEEDS TO DEMONSTRATE MOST OF THE BEHAVIOR OF THE FINAL PRODUCT

WHAT IS A PROTOTYPE?

- WHAT ARE THE GOALS FOR A PROTOTYPE?
- WHAT ISSUES ARE INVOLVED?
- HOW DOES IT FIT INTO THE SOFTWARE LIFE CYCLE?
- HOW DO YOU USE PROTOTYPES?
- WHAT TOOLS CAN BE USED TO:
  - DESIGN PROTOTYPES?
  - BUILD PROTOTYPES?
  - EVALUATE PROTOTYPES?
- DOES IT FIT INTO THE NASA ENVIRONMENT?

WHAT IS A PROTOTYPE?

- "QUICK AND DIRTY" "THROW AWAY" FOR EVALUATION
- SUBSET IMPLEMENTATION
- HOW DIFFERS FROM "INCREMENTAL DEVELOPMENT?"

LANGUAGE LEVEL?

- "LOW" (FORTRAN, PL/I, PASCAL)
- "HIGH" (SETL, SNOBOL4)
- "VERY HIGH" (SPECIFICATION LANGUAGES—GIST)

M. Zelkowitz
U. of M.

NOW PROTOTYPING A RESEARCH ISSUE——

— PROTOTYPE BY SUPERCODERS

— DEVELOPMENT BY MERE MORTALS

1. WHAT EFFECT ON DEVELOPMENT OF TECHNIQUES?

2. WHAT WILL HAPPEN WHEN MERE MOTALS START TO PROTOTYPE?

IS NOT REALLY ADDRESSED

YET—MEASUREMENT

- **PROTOTYPE USED FOR EVALUATION, BUT *HOW* EVALUATED?**
- **USER "SATISFACTION", "USER FRIENDLY"**
- **PERFORMANCE**
- **COSTS**
- **NEED MODELS OF PROTOTYPING AND PROBES CAN BE ADDED TO PROJECTS TO PERFORM EVALUATION**

AREAS OF DISCUSSION

- PROFILES
- MODELS
- METHODOLOGIES

PROFILES

- LACK OF KNOWLEDGE ABOUT CHARACTERISTICS OF A PROTOTYPE
- WHAT IS REASONABLE COST RELATIVE TO FULL DEVELOPMENT?
- WHAT LEVEL OF RELIABILITY SHOULD BE ACHIEVED?
- WHAT LEVEL OF FUNCTIONALITY IS DESIRED?

NEED TO COLLECT DATA TO CHARACTERIZE THIS TYPE OF DEVELOPMENT

MODELS

- LIFE CYCLE MODELS
- ERROR MODELS
- COST MODELS

NEED TO COLLECT DATA TO GENERATE VARIOUS MODELS

AND TEST EXISTING MODELS ON PROTOTYPES

METHODOLOGIES

- HOW TO BUILD A PROTOTYPE
- HOW TO EVALUATE A PROTOTYPE
- HOW TO USE PROTOTYPE TO BUILD FULL IMPLEMENTATION

FLIGHT DYNAMICS ANALYSIS SYSTEM

CURRENT METHOD: (E.G., TO TEST NEW ORBIT CALCULATIONS):

- ACCESS FORTRAN SOURCE LIBRARY
- MODIFY PROPER SUBROUTINE
- RECOMPILE AND BUILD NEW LOAD MODULE
- TEST NEW ALGORITHM
- RUN EXPERIMENT

---| THUS NEED DETAILED KNOWLEDGE OF SYSTEM

FDAS:

- ENTER FDAS
- FDASACCESSES DATA BASE AND ASKS FOR TASK
- EXPERIMENTER SPECIFIES CHANGE
- FDAS RECOMPILES FORTRAN SOURCE AND BUILDS NEW LOAD MODULE
- RUN EXPERIMENT

---| LESS DETAILED KNOWLEDGE NEEDED OF SOURCE PROGRAM
AND LESS TIME NEEDED TO RUN EXPERIMENT

FACTORS IN SOFTWARE DEVELOPMENT

	GROUND SUPPORT SOFTWARE
REQUIREMENTS	KNOWN
SIZE	KNOWN
EXECUTION CHARACTERISTICS	KNOWN
USER INTERFACE	KNOWN
ALGORITHM DESIGN	KNOWN
COST	KNOWN

FACTORS IN SOFTWARE DEVELOPMENT

	NEW DEVELOPMENT
REQUIREMENTS	?
SIZE	?
EXECUTION CHARACTERISTICS	?
USER INTERFACE	?
ALGORITHM DESIGN	?
COST	?

PROTOTYPE STRATEGY

- DEFINE A SUBSET OF THE REQUIREMENTS OF A NEW DEVELOPMENT
- BUILD A PROTOTYPE TO THESE REQUIREMENTS
- INSTRUMENT THE PROTOTYPE TO COLLECT USAGE AND PERFORMANCE DATA
- COLLECT S.E.L. PROJECT DATA
- EVALUATE PROTOTYPE
- REDESIGN FEATURES THAT DO NOT MEET SPECIFICATIONS
- BUILD FULL IMPLEMENTATION
- EVALUATE EFFECTIVENESS OF PROTOTYPING ON FINAL PRODUCT:
    - CHEAPER?
    - RELIABILITY?
    - EFFICIENCY?
    - MAN/MACHINE INTERFACE?

AUTOMATED SCORES

- USAGE OF FEATURES
- TIMING DATA
- ERROR COUNTS
- HOW OFTEN PROTOTYPE IS BYPASSED

CONCLUSIONS

- GENERATE PROFILE OF PROTOTYPE DEVELOPMENT
- IS IT SUCCESSFUL IN NASA ENVIRONMENT?

COME BACK NEXT YEAR!!!

D5

N83 32361

PANEL #2

SOFTWARE TOOLS

J. Goguen/K. Levitt, SRI
I. Miyamoto, University of Maryland
P. Szulewski, Draper Labs

EXPERIENCES AND PERSPECTIVES WITH SRI'S TOOLS FOR SOFTWARE DESIGN AND VALIDATION

by Joseph Goguen and Karl N. Levitt
Computer Science Laboratory
SRI International
Menlo Park, CA 94025

For the past 10 years SRI has had a major research program concerned with program specification, design and verification. The product of this work has been an evolving methodology supported by specification languages and tools for reasoning about specifications. Among the most important tools are: syntax and type checkers; semantic checkers and theorem provers; interpreters for processing test data; and analyzers for proving particular properties of specifications (e.g., the absence of security violations). To evaluate this methodology, we have undertaken successful large scale applications to both fault tolerant computing and to secure computing. Our research is now evolving to an environment that can support the entire programming lifecycle. Among tools now under construction for this more comprehensive methodology are structured editors, pretty printers, program libraries, and program testing systems. We are also considering the use of graphics, e.g., pictures to display important properties of systems. This paper briefly describes the current methodology, with emphasis on the role of specifications in the design process, and presents our experience (and that of others who have used the techniques) on several significant projects.

We have found it useful to consider a spectrum of different specification languages, each most suitable for a different purpose. A major purpose of a specification language is to support the decomposition and testing of designs at an early stage, so as to forestall unnecessary effort at later stages. Sometimes it is only necessary to obtain a prototype system which demonstrates the feasability of some concept; in such a case, it would be desirable to directly execute its specification. In other cases, one wants to be able to easily verify some particular but subtle property of a system, such as its ability to recover from certain classes of faults; then one might want to structure the design to facilitate the proof. In other cases, one might want to use specifications for documentation, and thus maximize their understandability and flexibility. In still other cases, one might want to be able to change easily from one design to another closely related one for a slightly different application or context. The languages and environments that are best for one of these purposes will not necessary be the best for another, and we have found that there are interesting trade offs, for example, between the expressive power of a specification language and its intuitive simplicity.

Although not denying the utility of specifications, designers have in general been reluctant to write formal specifications. Perhaps the most compelling reasons for this have been the absence of a good specification language with tool support and the absence of examples that can serve as a

model of a "good" specification; a specification with too much detail is not worth the effort. Consequently, formal methods have only been seriously attempted for those systems where reliablity is vital. We see these methods as now becoming ready for a broader class of systems.

In support of our efforts, we are developing tools that include the following: the STP theorem prover and its associated Design Verification System (developed by Schwartz, Shostak, and Melliar-Smith); PHIL, a meta-programmable context sensitive structured editor (developed by Goguen and Lamport);

Pegasus, a system for support of graphical programming; and OBJ, an ultra high level programming language based on rewrite rules and abstract data types (developed by Goguen). We are also doing some related work on acquiring and expressing requirements, and on performance analysis.

We have had particular success with the specification and verification of two classes of systems for which reliability is vital: fault-tolerant systems for aircraft control and secure operating systems. For the former, we have developed a fault-tolerant computer called SIFT (Software Implemented Fault-Tolerance), and have verified that it is correct with respect to a reliability model. Several subtle bugs in our original software were uncovered in the process of specification and verification. The most significant was that the results of infrequently executed tasks were not voted on sufficiently often and, hence, were not adequately protected against faults.

For the secure systems work, we (in cooperation with Honeywell Systems and Research, Ford Aerospace, and several other companies) have worked on several secure operating systems, ranging from small guards and kernels to a full, general purpose operating system (PSOS -- Provably Secure Operating System). For PSOS, in particular, the salutary effects of producing formal specifications were:

- A clean decomposition of the system into modules that are largely independent
- Minimization of the total number of modules through the the identification of multipurpose, parameterized modules
- A clean user interface
- A portable design in that each level in the hierarchy provides an interface independent of how it is implemented
- Identification of easily-formulated properties that were used as the basis in proving a design to be secure.

THE VIEWGRAPH MATERIALS

for the

J. GOGUEN/K. LEVITT PRESENTATION FOLLOW

WORK AT SRI INTERNATIONAL ON SOFTWARE SPECIFICATION AND REQUIREMENTS

JOSEPH GOGUEN
KARL N. LEVITT

COMPUTER SCIENCE LABORATORY
SRI INTERNATIONAL
MENLO PARK, CA

OUR MESSAGE

- A "new" paradigm for software development is gaining acceptance

- FORMAL (i.e., precise) REQUIREMENTS and SPECIFICATIONS are now possible for most systems

- Experimental languages and tools for analyzing specifications and requirements are available, e.g, SRI's Hierarchical Development Methodology (HDM) and specification languages SPECIAL and OBJ

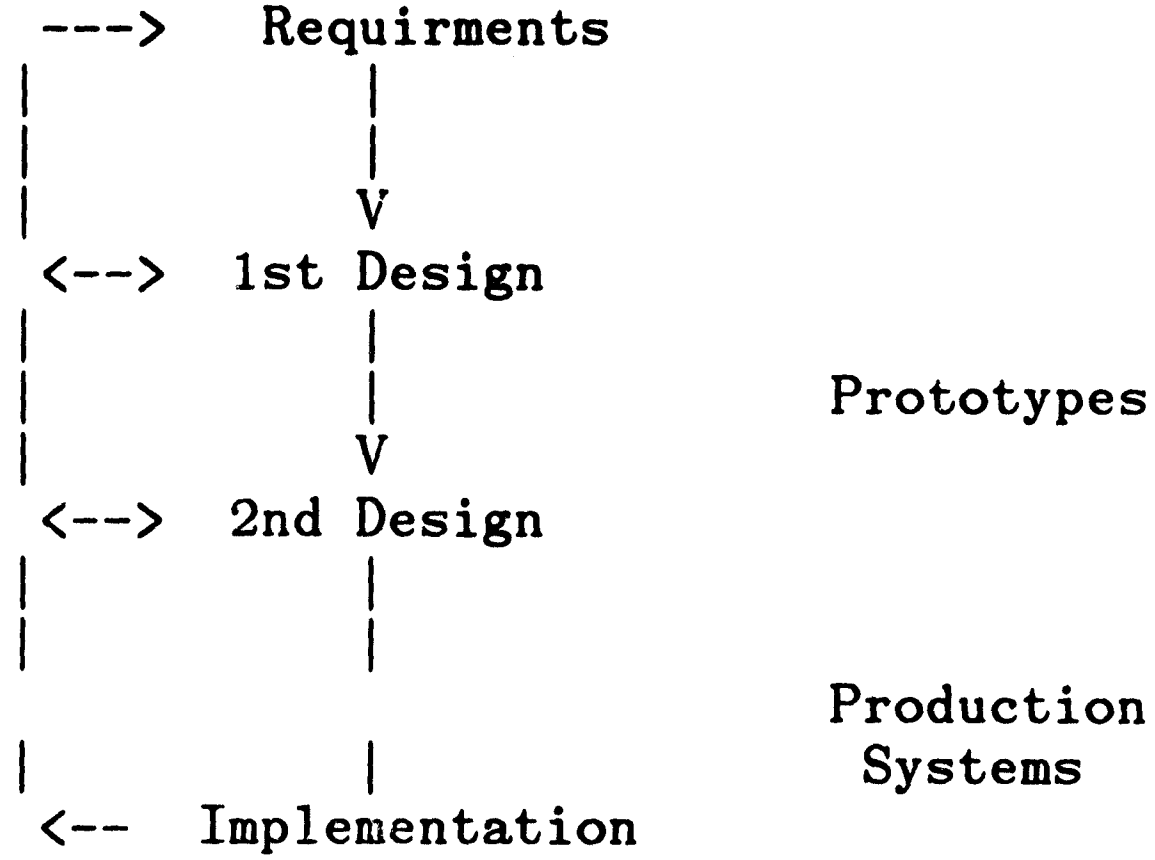
- Experiences with these techniques have been positive

  * SIFT (Software Implemented Fault Tolerance) ultrareliable flight-control computer

  * PSOS (Provably Secure Operating System)

- These techniques give promise of reducing lifecycle cost

PREFERRED APPROACH TO SOFTWARE DEVELOPMENT



ACTIVITIES AT EACH STAGE

Formal specification -- functional behavior	Supported in HDM and OBJ
Verification of specs	Supported in HDM
Testing of executable specs -- with real and symbolic data	Supported in OBJ
Interstage consistency (including design and code verification)	Supported in HDM
Pictorial descriptions of specs and code	In progress

APPROACHES TO INTERSTAGE REFINEMENT

- Vertical refinement -- Hierarchical decomposition using Abstract Data Types

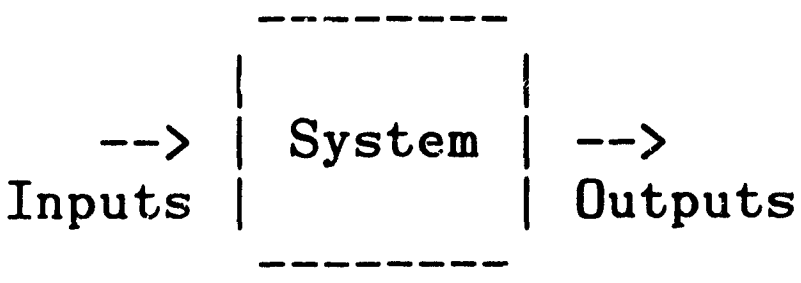
- Horizontal refinement -- Building a module out of existing modules

- Program transformation -- Improving the performance of a program while preserving its functional behavior

WHAT IS A SPECIFICATION

A specification is the DEFINING statement of a system's BEHAVIOR

It should resolve UNAMBIGUOUSLY questions about how the system should resolve in ANY situation



A spec is a BLACK-BOX Description

UNAMBIGUOUS => specs are FORMAL

QUALITIES OF A "GOOD" SPECIFICATION

- Concise
- Easy to produce (compared with an implementation)
- Readable
- Executable (in support of testing)
- Support automated reasoning (e.g., verification)
- Allow for performance analysis and/or simulation

FEATURES OF A SPECIFICATION LANGUAGE

- Allow specification just in terms of "callable" functions. E.g., a "file" system is definable in terms of

  CreateFile, OpenFile, CloseFile, WriteFile, ReadFile, MovePointer

  An OBJ specification consists of equations e.g.,

```
ReadFile(WriteFile(CreateFile(), val)
      = val
```

- Allow specification in terms of abstract (i.e., high-level data structures

  An HDM specification would represent a "file" in terms of a semi-infinite array (FileVal) and a pointer (FilePointer)

```
WriteFile(val)
 EXCEPTION: FileFull
 EFFECTS:
  'FilePointer = FilePointer + 1
  'FileVal('FilePointer()) = val
```

FEATURES (cont.)

- PARAMETERIZATION, i.e., using a library of previous developed specifications

  a "secure" file could be specified as

  SecureFile(Contents, SecurityLevel)

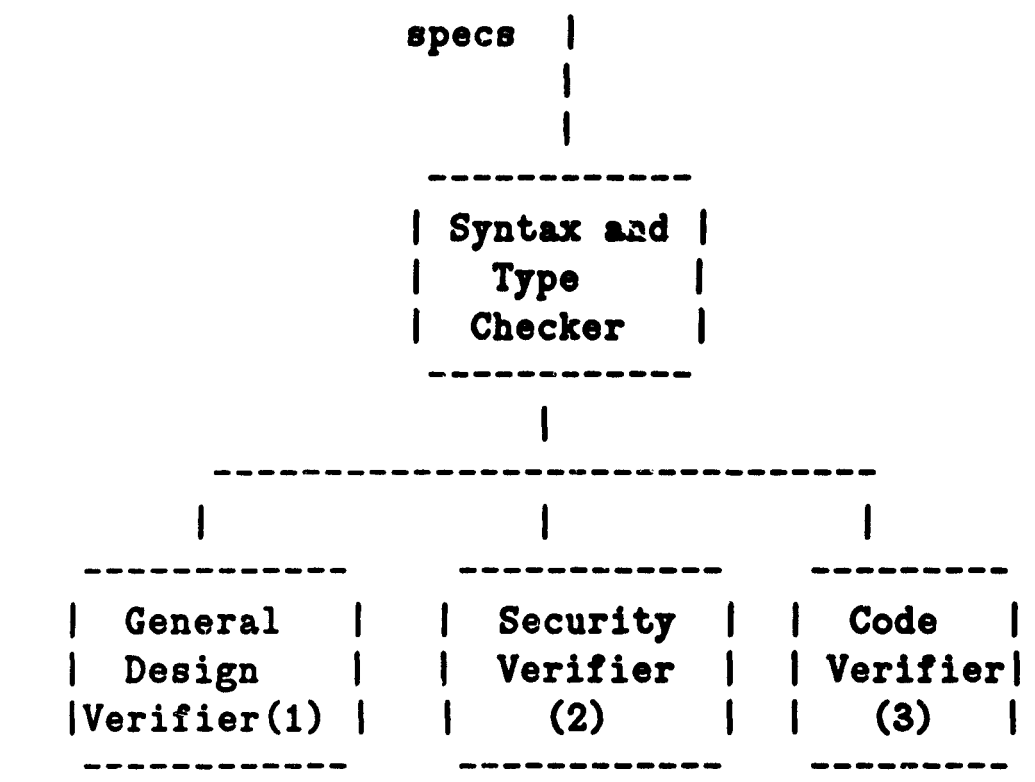
  where:
  Contents is any type
  SecurityLevel is "Partially Ordered Set"

- Logical and Set statements (including infinite sets)

  Finding an element val in a file:
  EXISTS i : FileVal(i) = val

  Number of appearances of element val in file:
  CARDINALITY({ i | FileVal(i) = val })

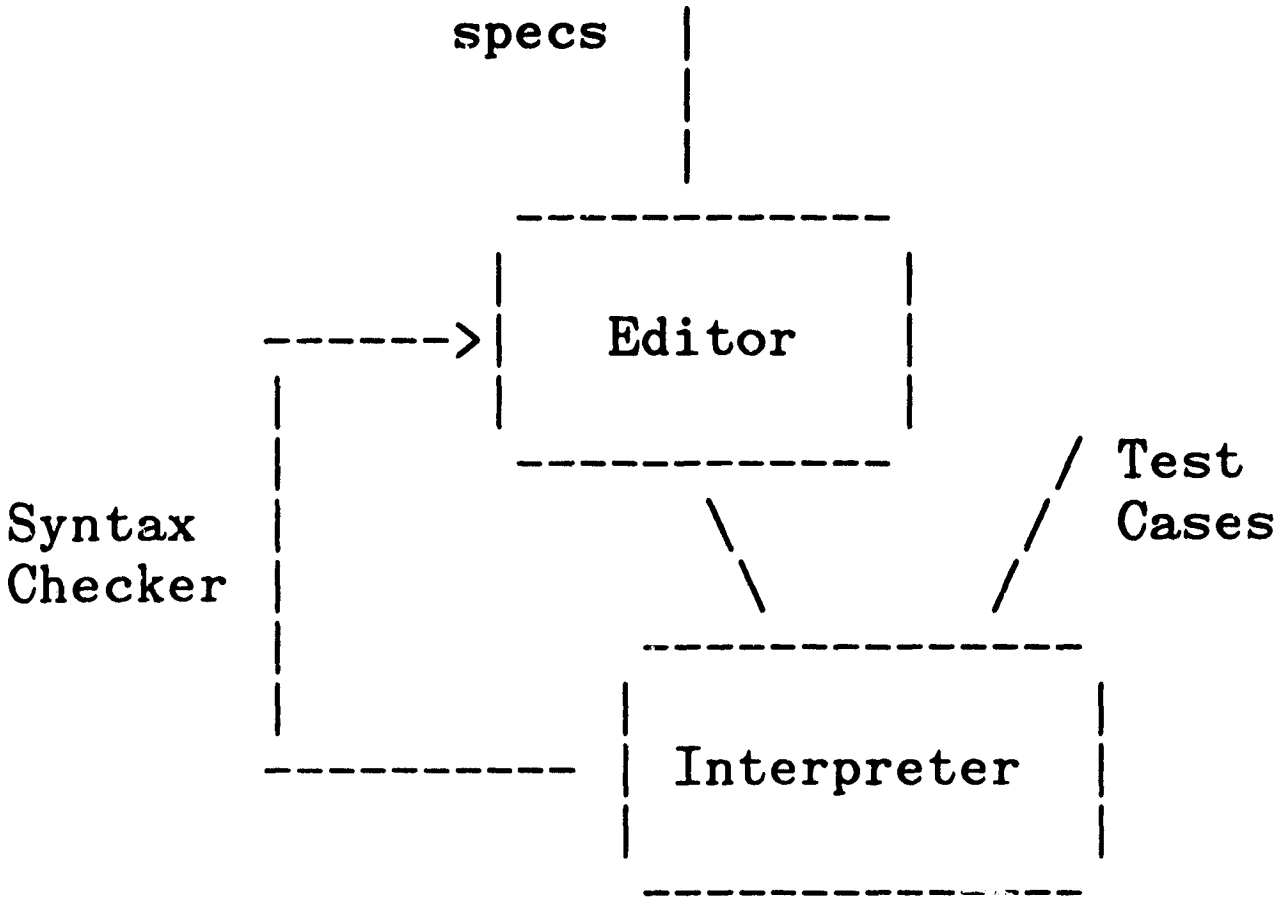
TOOLS IN SUPPORT OF HDM AND SPECIAL



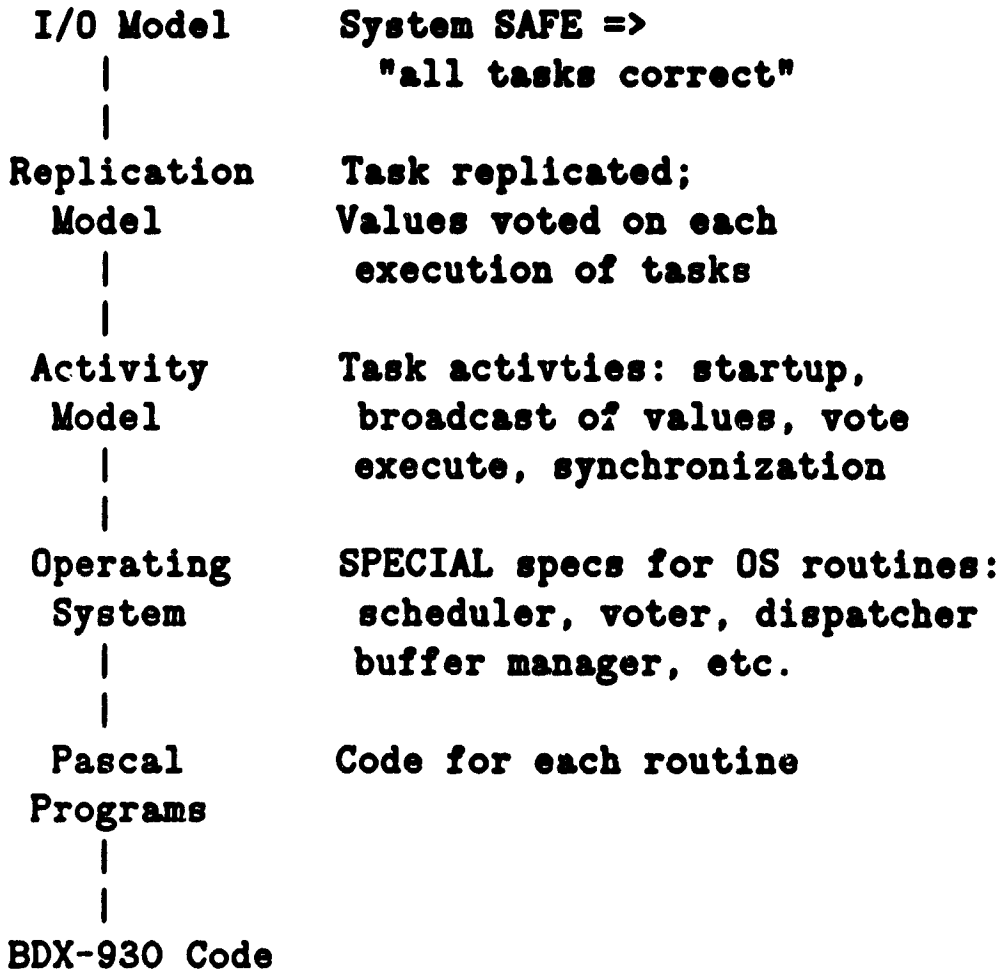
Notes:

1. Verifies properties of spec, e.g, "File will never overflow"
2. Checks for information flows in violation with Multi-Level Security Model
3. Languages supported: Pascal, Jovial, Fortran 77

TOOLS IN SUPPORT OF OBJ



REFINEMENTS FOR SIFT ULTRARELIABLE COMPUTER



EXPERIENCE WITH SRI's FORMAL TECHNIQUES

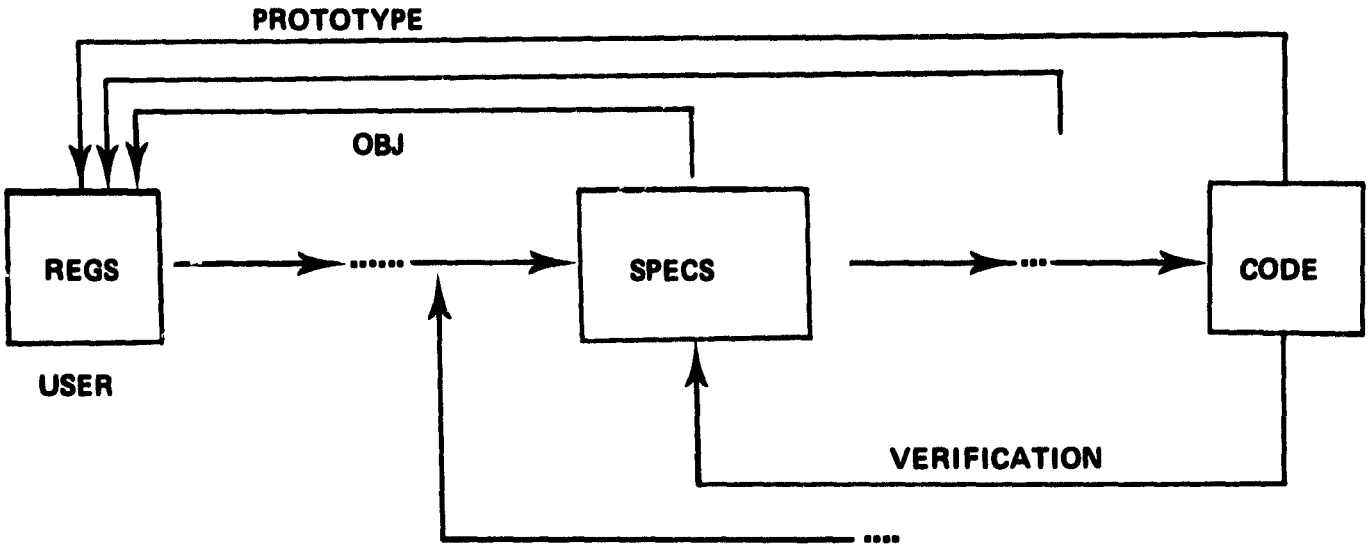
Organization	System	Specs	Design Proof	Code Proof
SRI	SIFT	x	x	x
	PSOS	x	x	
	Real-time OS	x		
Ford Aerospace	KSOS-11	x	x	
Honeywell	SCOMP	x	x	
Sytek	SACDIN	x	x	
Merdan	Secure msg system	x	x	

PSOS DESIGN HIERARCHY

LEVEL	PSOS ABSTRACTION OR FUNCTION
16	USER REQUEST INTERPRETER *
15	USER ENVIRONMENTS AND NAME SPACES *
14	USER INPUT-OUTPUT *
13	PROCEDURE RECORDS *
12	USER PROCESSES * AND VISIBLE INPUT-OUTPUT *
11	CREATION AND DELETION OF USER OBJECTS *
10	DIRECTORIES (*)[C11]
9	EXTENDED TYPES (*)[C11]
8	SEGMENTATION AND WINDOWS (*)[C11]
7	PAGING [8]
6	SYSTEM PROCESSES AND INPUT-OUTPUT [12]
5	PRIMITIVE INPUT/OUTPUT [6]
4	ARITHMETIC AND OTHER BASIC OPERATIONS *
3	CLOCKS [6]
2	INTERRUPTS [6]
1	REGISTERS (*) AND ADDRESSABLE MEMORY [7]
0	CAPABILITIES *

* = MODULE FUNCTIONS VISIBLE AT USER INTERFACE.
(*) = MODULE PARTIALLY VISIBLE AT USER INTERFACE.
[I] = MODULE HIDDEN BY LEVEL I.
[C11] = CREATION/DELETION ONLY HIDDEN BY LEVEL 11.

S/W Eng Methodology



PROTOTYPING: Feedback to user is a fuzzy concept

EXAMPLE. Use of scenerios

ALSO need feedback to the designer/coder

e.g., performance models

TOPICS: Early in process

This roughly corresponds to level of abstraction

GENERAL MOTIVATION

TO provide a precise scientific way

TO discover

a) What users want or need

(requirements)

b) What "linguistic structures"* work best for a given purpose

(user interface design)

c) What is really going on in a given social context

(social system analysis)

---

* may be graphical, textual, speech, or mixed media; all are "linguistic" in the sense of being hierarchically structured into atoms, phrases, and discourse units.

REQUIREMENTS

Two major components:

1. How the client will use the system.

   information flow at the interface, inside the system, and in the client's organization.

2. Client's criteria for evaluation of the system.

   a hierarchy of values: may be subjective factors and organizational factors, as well as objective and individual factors.

These lead to two representation systems.

1. Abstract Data Flow Diagrams

2. Value System Trees

Note that both are graphical in nature.

ABSTRACT INFORMATION FLOW

A. MOTIVATION

We want to characterize information by its use and intention (social meaning), not by its physical representation.

vs. operations research

This can be done if we look at the information from the viewpoint of those who use it.

Such information is available in the users' language.

B. DATA FLOW DIAGRAMS

Graphs, with "files," which represent some type of data, generally structured; and "actions," which are operations on that data.

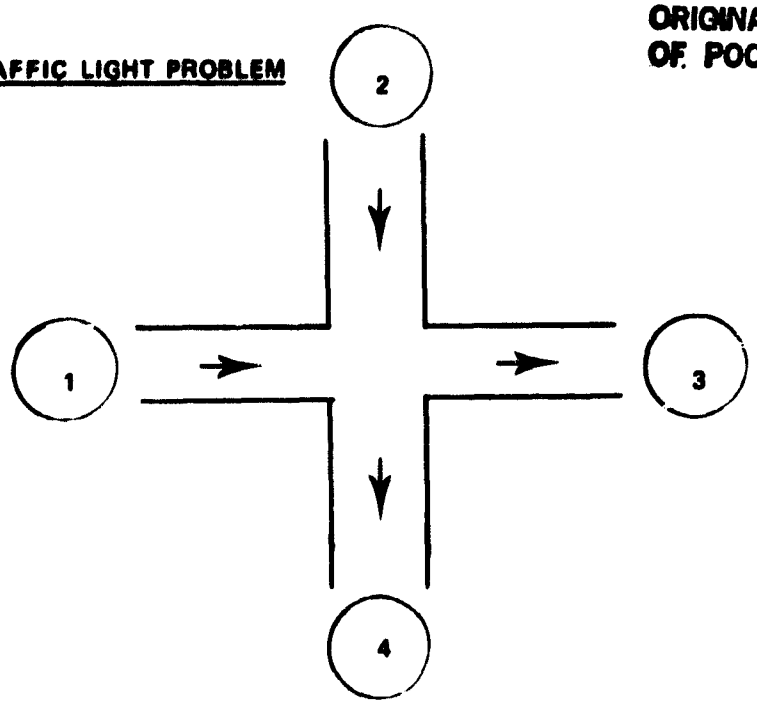
We can have both iteration and recursion in DFDs.

Also hierarchical structure.

C. ABSTRACT DATA FLOW DIAGRAMS

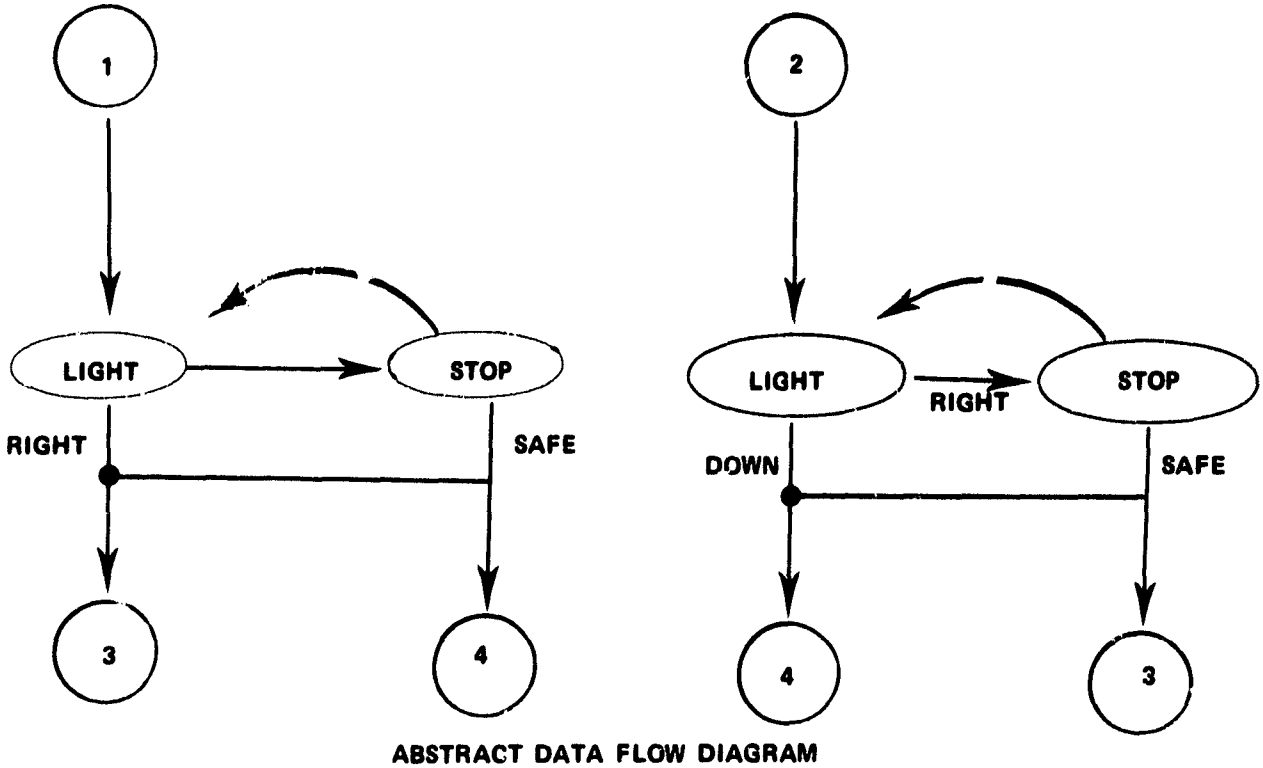
"Abstract" means independent of representation data characterized by relations among op's on it.

TRAFFIC LIGHT PROBLEM



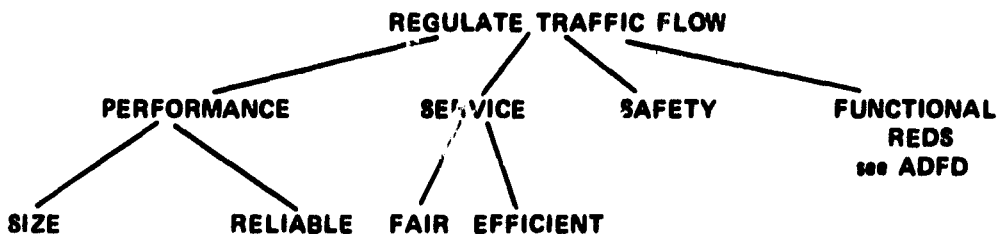
INTERSECTION

CAR = ADT



ABSTRACT DATA FLOW DIAGRAM

"abstract car flow processor"
can be compiled into a simulation



VALUE SYSTEM TREE

Can be used to organize:

- Management effort
- Organizational structure
- Accounting
- Structured walkthroughs
- Acceptance tests
- Redesign criteria
- . . .

Natural visualization

Can be used to compile tools for later phases.

C - 3

D6

USER INTERFACE DESIGN OF SOFTWARE TOOL SYSTEM AS A TECHNOLOGY TRANSFER VEHICLE

Isao Miyamoto
Department of Mathematics and Computer Science
University of Maryland, Baltimore County
Catonsville, MD 21228

ABSTRACT

The paper introduces design considerations of an on-going research project for developing an effective and easy-to-use tool system that supports entire maintenance phases. The primary focus is the design of an "intelligent" user interface mechanism. By analyzing why existing tools and tool systems are not used very effectively, we can define users requirements for the user interface mechanisms, specify design criteria of user interface functions, and introduce some features of the implementation. Because this project is still in process, intermediate evaluation and expected effectiveness are discussed. The author believes that only a well-designed tool system can be a powerful software engineering technology transfer vehicle.

1. INTRODUCTION

The role of the software system is extremely important in a computer-based system. The technology to develop and maintain quality software is the key to the advancement of computer applications; such technology is called software engineering
We have surveyed current techniques, methodologies, and tools (or tool systems) for producing high quality software [1]. The most serious finding is that although many techniques, methodologies, and software tools are available, they are not used very much or very effectively in real software production environments [2]. Sometimes, programmers do not know what items are available or how to use them. Sometime, their productivity and quality of their software fail to improve anyway. Later we will discuss some reasons for the failures.

The author has experience in the development of a large-scale integrated tool system. This project was carried out in the author's former company from 1976 to 1978. We tried to develop a software support system named Software Development & Maintenance Support System (SDMSS)[6] that was supposed to cover the entire software life cycle. Although we had developed some parts of the system, I frankly think we failed to develop an easy-to-use and effective tool system. We did not spend enough time designing the framework of the system, such as maintainability, portability, database, command language, graphics capability, etc. We simply tried to integrate many attractive ideas. We required a very large host computer, much programming effort, many resources to execute this system, etc. We did not have any clear methodology for using all of the functions of the system. We learned many lessons from the failure of this project

In addition to this experience, the author has promoted modern software engineering techniques in the software industries. Through this type of professional development, the author made valuable findings about the issues of technology transfer. To summarize, transfer of technology is very difficult if we lack tools that realize and support the proposed methodology or technique.
From those experiences, we discovered why it is difficult to transfer software engineering technology from the research environment to the production environment. The production environment is in great need of these new ideas. We also realize why existing individual tools and tool systems are not used very much or very effectively although they were developed to be used frequently. Some of the problems come from management, some come from human factors, and many are associated with the tool or tool system itself.
However, many of those reasons may be integrated as a "technology transfer problem." We would like to introduce some ideas for the transfer of software engineering tools.

1.1 Why software tools are not used.

Our survey [3] and some other surveys [4,5] indicate that we have many individual tools and several tool systems. However, almost none of these is used effectively.
For individual tools, some of the major reasons are as follows.

1) Most tools do not have a clearly defined methodology, and only the program code is available. Rarely is a user's guide available.

2) Most tools have not verified their economic effectiveness.

3) Because of the difficulty in defining criteria for evaluating the quality and effectiveness of software tools, many tools have not been tested by users.

4) Many tools are not evaluated at all.

5) Some tools have been evaluated, but they are clearly not cost-effective.

6) It is very hard to use or describe some tools.

7) Documentation (user's manual, design specification, maintenance manual, etc.) is poor. Sometimes there is no available documentation.

8) Tools assume many predetermined environmental conditions which are not documented. Most of the time, these conditions do not match the real conditions of the users.

9) Usability of tools is very poor because of lack of proper methodology

10) Sometimes the tool itself does not properly support user activity because of poor understanding of software production process models.

11) The reliability of the algorithms, the quality of the implementation, and the efficiency of the tool are not sufficient for the user.

12) Many individual tools are not designed to have common input/output formats

13) Users strongly resist tools that were designed at other organizations.

14) Special-purpose tools service a very small audience.

15) Programmers generally resist new or foreign languages and tools. Expert programmers are the most resistant, as they are the most conservative.

16) If the development group has a bad reputation, most programmers do not want to use the products.

17) Sometimes, the particular tools have a bad reputation.

18) Sometimes tools do not fit the existing working criteria.

19) Many tools do not have extendability or modifiability to accommodate each user's environment

20) The maintenance of the tool itself has not been taken into account properly; and the quality and functionality of the tool become ineffective over time.

21) The portability of the tools is very poor.

Several points represent the problem of designing our tools to improve the situation. For example, reasons 1, 6, 7, 8, 9, 12, 19, 20, and 21, all depend on design or on support methodology to apply a tool's capabilities to the user's proper production activites.

2 Why tool systems are not used

Tool systems are collections of many individual tools. There are two types of tool system: heterogeneous and homogeneous.
The first type of tool system integrates different types of tools and supports no common methodology for using the component tools. The second type also integrates individual tools but supports some common methodology for using the component tools. UNIX is representative of the first type, and SDMSS is representative of the second type [6].
Each type has both merits and demerits, neither is a perfect tool system.
Existing tool systems have the following major problems.

1) In general, tool systems have the potential to be bigger and bigger. To create and use a tool system requires a large memory space, many computer resources, a large database, a large-scale computer, sophisticated terminal devices, etc.

2) A particular tool system is very expensive to use.

3) The development cost of a tool system itself is extremely high.

4) Components of tool system are tightly integrated and so adding or deleting tool functions is quite difficult.

5) The maintenance of a tool system itself is tremendously expensive, in fact, sometimes it is impossible to maintain.

6) The input and output of the components are not uniform.

7) Many user interfaces of tool systems depend on the host operating system, and they are not easy to use.

8) Because many functions depend on the specific hardware or operating system, the portability of the tool system to other environments is very poor.

9) Very few tool systems are designed to support both expert programmers and novices.

10) Most tool systems are not designed to support groups of users.

11) Few tool systems are graphics-oriented, and so many users must use text type information.

12) Most tool systems do not have any global-level methodology, and are just a collection of individual tools.
Sometimes, tool systems enforce a very biased (e.g. improper, and always same) methodology to users.

13) Management of the activities done through the tool systems is not available.

14) It is difficult to cover the entire software life cycle because of the current level of sophistication of software technology.

A tool system has many problems beyond those of tools This is why very few existing tools or tool systems are used very much or very effectively. In order to increase software productivity and software quality [1] we must design and use support tools (or tool systems) that aid our software development and maintenance activities at least
Therefore a question we must answer is how to design effective and easy-to-use tools or tool systems
A tool or tool system can be a very powerful medium that transfers software engineering methodology from researchers to practioners in the real world Carefully designed tool systems can very effectively transfer technology We believe that tool design is only one strong mechanism to aid transfer of existing technology We also believe that tools must be easy-to-use and cost-effective to make people apply new software engineering technology

2 PROBLEMS

We are designing a rather ambitious tool system to support software maintenance activities which is called Pandora's Box [7] The tools are available individually now Taking into account previous major problems we have carefully defined our design criteria
Concerning the big scale of tool systems (Problem #1) the Pandora's Box is designed to have several subsystems which are independently executable Entire whole functionalities of tool system are going to be very large but each component is designed to be very compact and to be executable on a super micro-computer Therefore the usage cost is expected to be very low (Problem #2) These components and functionalities are designed to use available tool functions (e g full use of UNIX environment) Then we will avoid wasting much money duplicating those functions The structures of subsystems are to be modular and all necessary interactions are to be done via database (Problem #4) Then the system structure is very flexible and each function is designed to be rather small to increase maintainability of the tool system itself (Problem #5)
Interactions between tool functions are done by database (or f software knowledge base) and input-output formats are common as in UNIX (Problem #6) With UNIX environment as a host for this tool system the portability of the system is assured to some extent (Problem #8) This tool system is graphics-oriented Then users can use the graphics capabilities of a color graphics terminal and color x-y plotter In the system graphics are not secondary to or a substitute for text type commands The policy calls for graphics first and text next (Problem #11) The tool system is designed to keep all of the usage history and register individual scenarios Using a scenario system and a hierarchical menu system we can manage the user's activities and collect some management data (Problem #13) We have tried to apply the latest techniques to the design of Pandora's Box and we limit the usage of the system to certain phases of the software life cycle
We selected only already evaluated techniques and tools (Problem #14) The problems related with user interface and methodology (Problems #7 #9 #10 and #12) are described precisely in the next section

3 DESIGNING USER INTERFACE MECHANISM

We are design ng a software-maintenance support tool system named 'Pandora's Box" and would like to introduce some ideas from this project. Those ideas are related to the user interface design of this tool system. The user interface is designed to have two fundamental functions for users. One is a three-level menu hierarchy to serve different scenarios to various type of programmers, from expert programmers to novice programmers. Another function is the knowledge-base guidance mechanism for those users.

3.1 Basic requirements of user interface functions

We assumed three types of users: novice users, expert users, and frequent expert users. Each type of user has different requirements for the user interface functions.

For example, novices need (quoted from [9]):

- utmost in clarity and simplicity,
- small number of user commands,
- meaningful commands (not a single letter, and not with complex syntax),
- lucid error messages and help facilities, and
- reinforcement from success.

Novices may want computer-directed mode and system's "friendliness."
Infrequent expert users prefer:

- simple commands,
- meaningful commands,
- easy to remember operations, and
- prompting.

On the other hand, frequent expert users want:

- powerful commands, command strings, user-defined commands,
- minimal number of keystrokes,
- brief messages (with access to detail at request), and
- high speed interaction.

Experts demand user control and system's "intelligence."
In order to satisfy all user levels, how should we proceed?

We might do the following: 1) to expect a 'graceful evolution' of users themselves, 2) to apply 'information hiding' techniques to user interface mechanism, or 3) to have a hierarchical menu selection system with "intelligence" and "individual" scenario. In general, a menu selection user interface may give us:

- little training,
- little memorization,
- clear structure for user activity,
- ease in designing individual small tool functions, and
- simplicity in software structure.

But because of the predetermined entries in the menu, usage can be somehow restrictive. In order to design a good menu selection system, we need to make a big development effort.

3 2 Design criteria of hierarchical menu system

By taking into account the basic requirements of user interface functions and referring to the material [9] and borrowing some ideas, we have set up the following design criteria:

apply intelligent user guidance mechanism,
use small number of choices per screen,
consider semantic organisation and give title,
show hierarchy by graphic design,
permit simple back, left, right, up and down traversals in the menu hierarchy,
use proper combination of colors,
permit type-ahead,
put most important and frequent choices first,
begin choices with keyword, if possible, and
require an enter key or use light-pen mode consistently.

Some other considerations are:

display rate,
response time,
help/explain facilities,
short cuts/menu macro, and
human reaction to colors.

3 3 Some features of user interface

a hierarchical scenarios

The scenario hierarchy of Pandora's Box consists of three levels of menus. The top level is so-called "methodology-oriented menu" (or scenario), and this will provide users "how-to-maintain scenarios" which will guide users to do all of the necessary maintenance activities. Those activities include the detailed phase plans of each type of maintenance. The scenarios are prepared in flexible way for emergency maintenance, planned maintenance, deferred maintenance, and preventive maintenance. The work breakdown structures and necessary procedural steps are the elementary source of this level. When users interact with this scenario, users can get complete guidance as to how to maintain users programs and data sets without precise knowledge about maintenance activities. The users do not need any written guideline to maintain their software, they need only follow.
The second level menu is "how-to-select proper tool functions menu" to do necessary action guided from the top level menu. The elementary information of this level is a list of tool functions provided by the Pandora's Box. The tool system will be expanded to contain all of the functions necessary to do all of the maintenance activities from maintenance requirements analysis to validation of maintained software. The menu at this level is constructed based on an activity-tools function] matrix. The third level menu contains the information about "how-to-use a particular tools function". This level gives users the exact information about the user commands to execute a particular tool function.

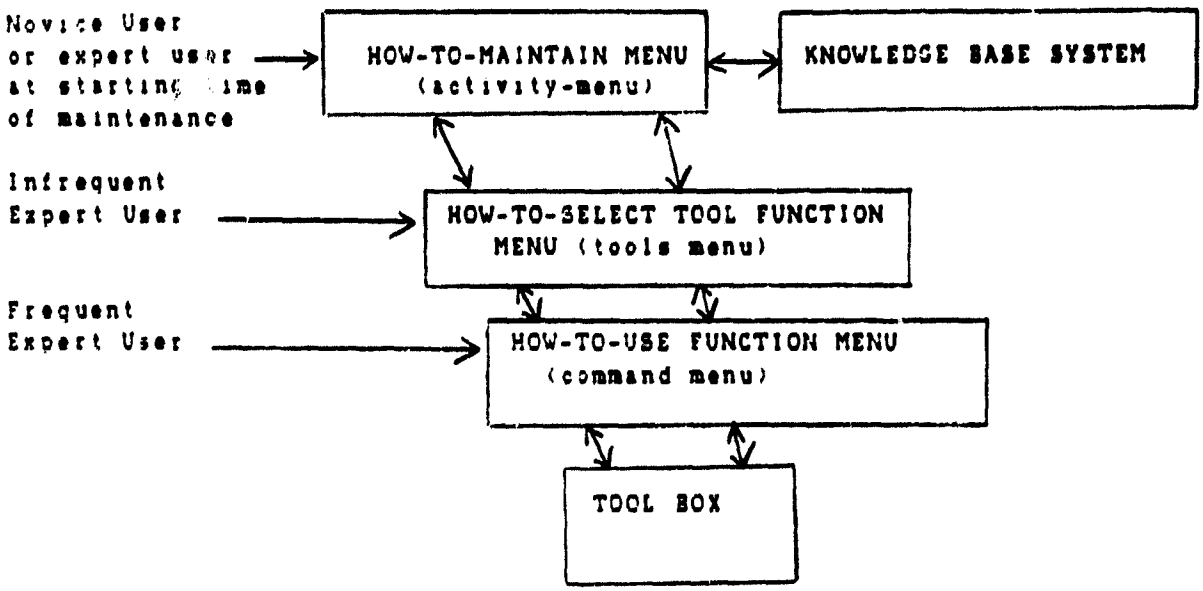


FIG. 1 Hierarchical Menu System

Figure 1 is a representation of this menu hierarchy. Users can access the Pandora's Box in any way from the top (in this case, users will be guided smoothly to next level menu and finally guided to command-level menu,) to the lowest level of hierarchy to achieve some particular maintenance activity. The system will record the histories of activity profile use for each user; and so the system can provide the best scenario to each user individually when users access the system the next time. The system will guide users by scanning the menu hierarchy up and down. The top level menu provides users with the methodology to maintain software. Users do not need a maintenance guidebook and users manual of the tool system itself any more.
The system will guide users and provide necessary information and functions to do the necessary activities.

b. Knowledge-base guidance

When the system guides users, the system refers to a knowledge base that contains software error information and maintenance pattern information. The knowledge base contains exactly two types of error information; one is the general tendency type error occurrence distribution, the other is the error history of each user collected during their use of Pandora's Box. The latter type of error information is analyzed according to the target program and individual user.

Some basic ideas of the error information collection mechanism within the automated tool system are given in the previous article [6].

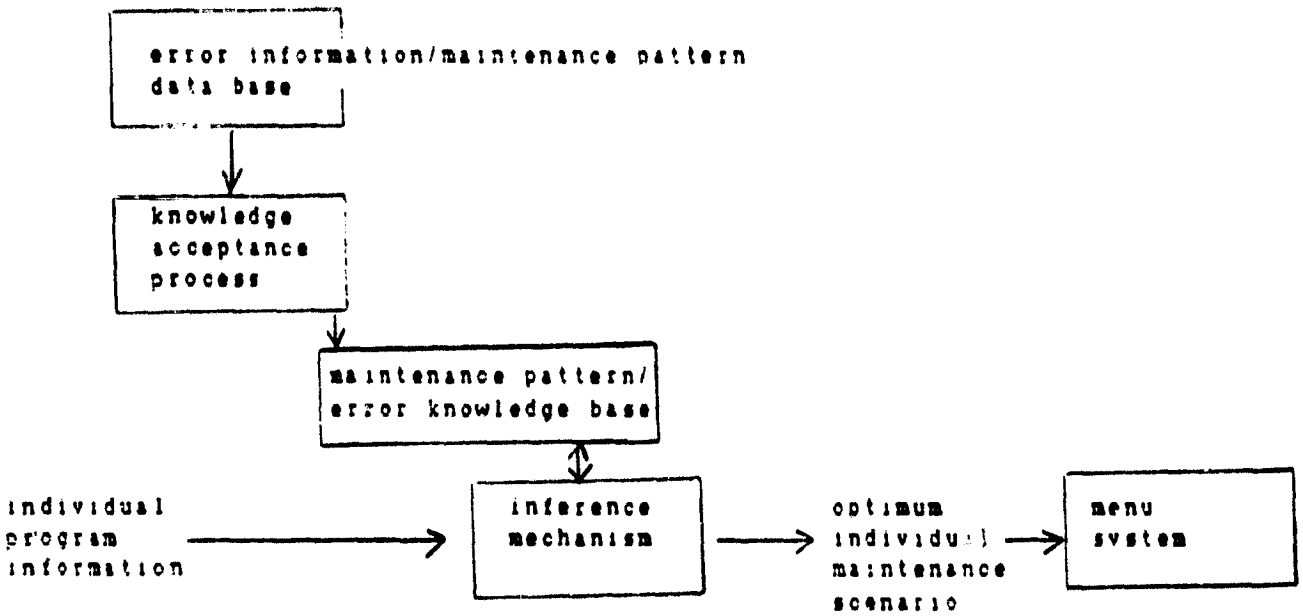


FIG.2 Knowledge Base System

In maintenance phase in general, especially in a case of corrective maintenance, to test modified programs in an efficient way is most important and necessary. As emphasised in [1], there are rather clear relationships between testing techniques and error types. To detect a particular type of error we need some specific techniques. We examined these relationships and made up testing techniques-error type matrix in the knowledge base system [7].

Referring to error information, we can get the information of the general tendency of error occurrence distribution, and by referring to the user's individual history, we may adjust this distribution to an individual user-oriented one. Based on this knowledge, at the time when the user signs on to Pandora's Box, we can provide the optimum individual maintenance scenario. This scenario is based on a prioritized menu so that the user can continue his/her most necessary and effective activities.

4 REMARKS

A technology transfer problem is not easy, because it is related to the education, training, techniques, methods, supporting tools, management organization, and human factors. Also, we don't know yet what should be transferred. Unless we know it, we can't discuss how we should do technology transfer. This may cause severe questions like, "what is a really useful software engineering technology to be transferred?", or "from whom to whom?" Beside discussions on the desk, we must take some approaches to improve the situation of existing tool usage. We hope that some of our ideas on the design of user interface for tool systems may show some possible direction.

Finally, "friendly" and "intelligent" user interface mechanism of well-designed tool systems could be a powerful technology transfer vehicle.

5 Acknowledgement

The author would like to express his special thanks to Drs. Victor Basili, Ben Shneiderman, Kouichi Kishida, and to his research associates for their advices and support.

REFERENCES

[1] I Miyamoto, "High quality software production techniques", TBS Pub Co., 1982, Tokyo, Japan

[2] I Miyamoto, "Management of software maintenance (No.5)", bit, Sept 1982, Kyoritsu Pub., Tokyo, Japan

[3] I. Miyamoto, "Management of software maintenance (No.4)", bit, Aug. 1982, Kyoritsu Pub., Tokyo, Japan

[4] Reifer Consultant, "Software Tools Directory"

[5] NBS, "Software Tools Directory", Oct. 1980

[6] I Miyamoto, "Reliability Evaluation and Management for An Entire Software Life Cycle", The 2nd Software Life Cycle Management Workshop, 1978

[7] I. Miyamoto, et al, "Conceptual design of Pandora's Box", to be appeared

[8] I. Miyamoto, "Management of software maintenance (No.3), bit, July 1982, Kyoritsu Pub., Tokyo, Japan

[9] Ben Shneiderman, Lecture Note of Software Engineering Seminar, Oct 1982, SRA International Inc.

THE VIEWGRAPH MATERIALS

for the

I. MIYAMOTO PRESENTATION WERE INCORPORATED IN THE PAPER.

DESIGN AIDS FOR REAL-TIME SYSTEMS (DARTS)

Paul A. Szulewski
The Charles Stark Draper Laboratory, Inc.
Cambridge, Massachusetts, 02139

Abstract

Introduction

Design-Aids for Real-Time Systems (DARTS) is a tool that assists in defining embedded computer systems through tree-structured graphics, military standard documentation support, and various analyses including automated Software Science [1] parameter counting and metrics calculation. These analyses provide both static and dynamic design quality feedback which can potentially aid in producing efficient, high-quality software systems.

DARTS Overview

DARTS uses a mix of hierarchy, control and communications primitives and data structures to represent real-time systems. Requirements are expressed as a functional hierarchy and designs as a tree-structured hierarchy of communicating processes. This hierarchical structure provides two distinct advantages, the system can be viewed at different levels of detail as required and changes (e.g., subtree move and delete) can be easily implemented.

Although developed specifically to represent real-time interactions, DARTS can be used to define both real-time and non-real-time systems. Specific real-time capabilities include an ability to represent and model

(1) interactions between the computer system and external sensors and effectors,
(2) interactions between processors in a distributed system design, and
(3) interrupt processing and the flow-of-control in multi-programmed software designs.

Through a friendly, menu-oriented interface, a user can represent a system; perform data flow checking; generate simulations of the design for response time, throughput, and utilization; request a variety of data tables and graphical tree-structured output in various sizes; and calculate Software Science complexity measures.

DARTS User Interface

DARTS is implemented as a PL/I program on an Amdahl 470 V/8. A user is presented with a menu-driven, full-screen interface [2] which users with no prior computer background have found easy to learn and use. Through this interface, an analyst can build and maintain a library of DARTS data bases, generate both graphical and tabular output, and initiate various analysis functions.

DARTS Data Base

The DARTS data base is hierarchical, with records corresponding to each of the nodes in the DARTS tree. The records contain data pertaining to control flow, data flow, and relational information for the nodes in the tree. Various attributes can be associated with the nodes of the tree. Nodes can have names, input and output variable lists, free text descriptors, durations, and actual assignment statements to be executed during a simulation. Nodes can also have predicates that determine the flow of control at branch points. Durations can be deterministic or can be given as random distributions. DARTS processes can be assigned priorities to allow one process to interrupt another. Thus, interrupt structures and preemption can be explicitly specified and modeled.

Data Flow Checking

Data flow consistency checking verifies that variables are produced before they are referenced and referenced after they are produced. Documentation outputs currently consist of a data base listing, the DARTS tree, a data-flow table showing data producer/consumer relationships for the nodes in the tree, data set/use tables, and module tables. These graphical and tabular outputs are embedded easily into word-processing files for automatic specification generation.

Automatic Simulation

A simulation capability [3] is available to provide estimates of performance factors, using a simulation language developed at the University of Birmingham, the Extended Control and Simulation Language (ECSL). A translator automatically converts a DARTS representation into an ECSL program. Statistics on performance factors such as response time, down time, utilization, and throughput are automatically collected and maintained by the DARTS/ECSL system. These statistics can be displayed in histogram formats for analysis.

Software Quality Metrics

An experimental metric of software design quality is among the design feedback analysis features in DARTS. These metrics, based on Software Science [1], are useful in assessing the quality of competing software designs as well as being predictors of other software planning parameters (e.g., size, effort, project duration, and number of modules).

Prior research [4,5] has shown that it is possible to identify and count Software Science parameters in software design media. Experimental data suggests that these metrics correlate with a subjective assessment of the criteria they were intended to measure.

References

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

[2] "Design-Aids for Real-Time Systems (DARTS): Users Guide," Version 3, CSDL-C-5441, The Charles Stark Draper Laboratory, Inc., January 12, 1982.

[3] Furtek, F.C., DeWolf, J.B., and Buchan, P., "DARTS: A Tool for Specification and Simulation of Real-Time Systems," Proceedings of the AIAA Computers in Aerospace III Conference, October 1981.

[4] Szulewski, P.A., Whitworth, M.H., Buchan, P., DeWolf, J.B., Quality Assurance Guidelines and Quality Metrics for Embedded Real-Time Software Designs, CSDL-R-1376, The Charles Stark Draper Laboratory, Inc., May 1980.

[5] Szulewski, P.A., Whitworth, M.H., Buchan, P., DeWolf, J.B., "The Measurement of Software Science Parameters in Software Designs," ACM SIGMETRICS Performance Evaluation Review, Vol. 10, No. 1, Spring 1981.

THE VIEWGRAPH MATERIALS

for the

P. SZULEWSKI PRESENTATION FOLLOW

The Charles Stark Draper Laboratory, Inc.

Cambridge, Massachusetts 02139

DESIGN-AIDS FOR REAL-TIME SYSTEMS

by

Paul A. Szulewski

Presented at the
Seventh Annual Software Engineering Workshop

December 1, 1982

Goddard Space Flight Center
Greenbelt, Maryland

8211C376-1

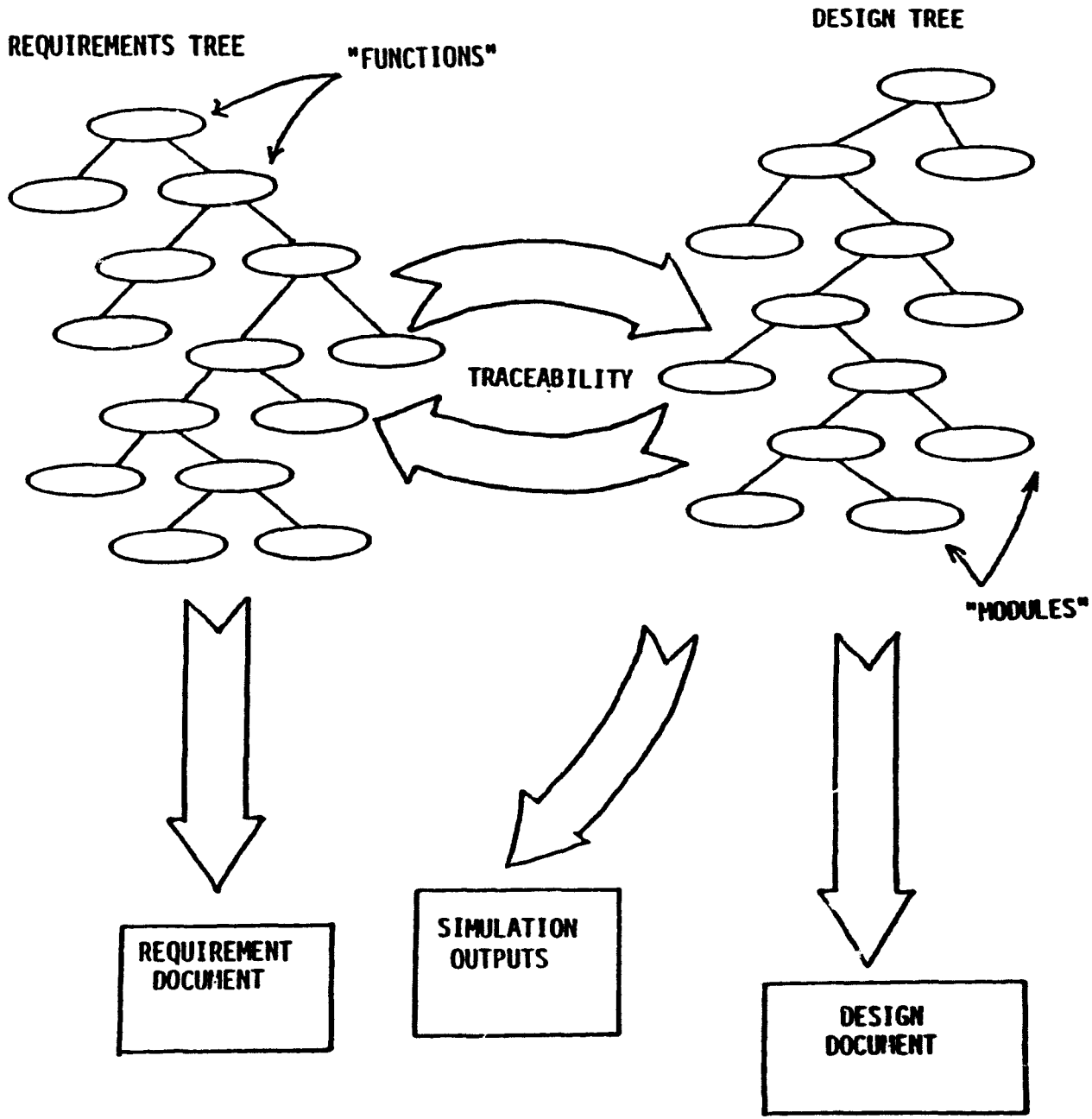
DARTS OVERVIEW

- What is DARTS?
  - An automated tool for the specification, simulation, and analysis of distributed, real-time systems
- What is its underlying model?
  - Hierarchical structure
  - Process oriented
- What features aid the designer?
  - Documentation in a variety of formats
  - Explicit control flow and data flow
  - Automatic simulation
  - Automatic software quality analysis
- What features aid management?
  - Concise and understandable documentation
  - Computerized data base

PROBLEM

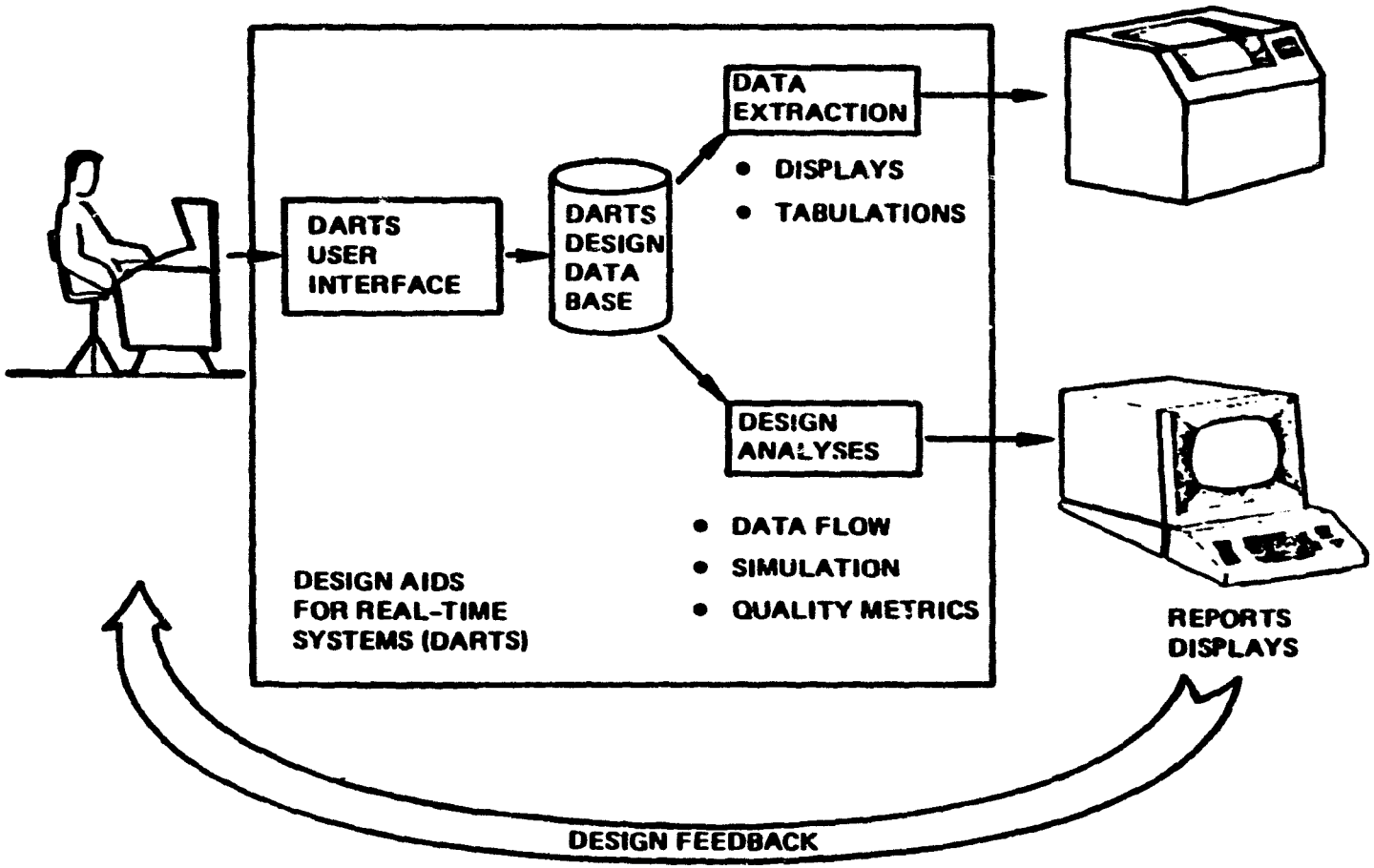
- **Defining requirements and preliminary designs**
  - **Crucial**
  - **But time consuming**
  - **Not Systematic**
- **Resulting deficiencies**
  - **Inadequate throughput/memory**
  - **Cost/time overruns**
  - **Reduced reliability, testability, maintainability**
  - **Project failure**

REQUIREMENTS/DESIGN METHODOLOGY



821\C376 7

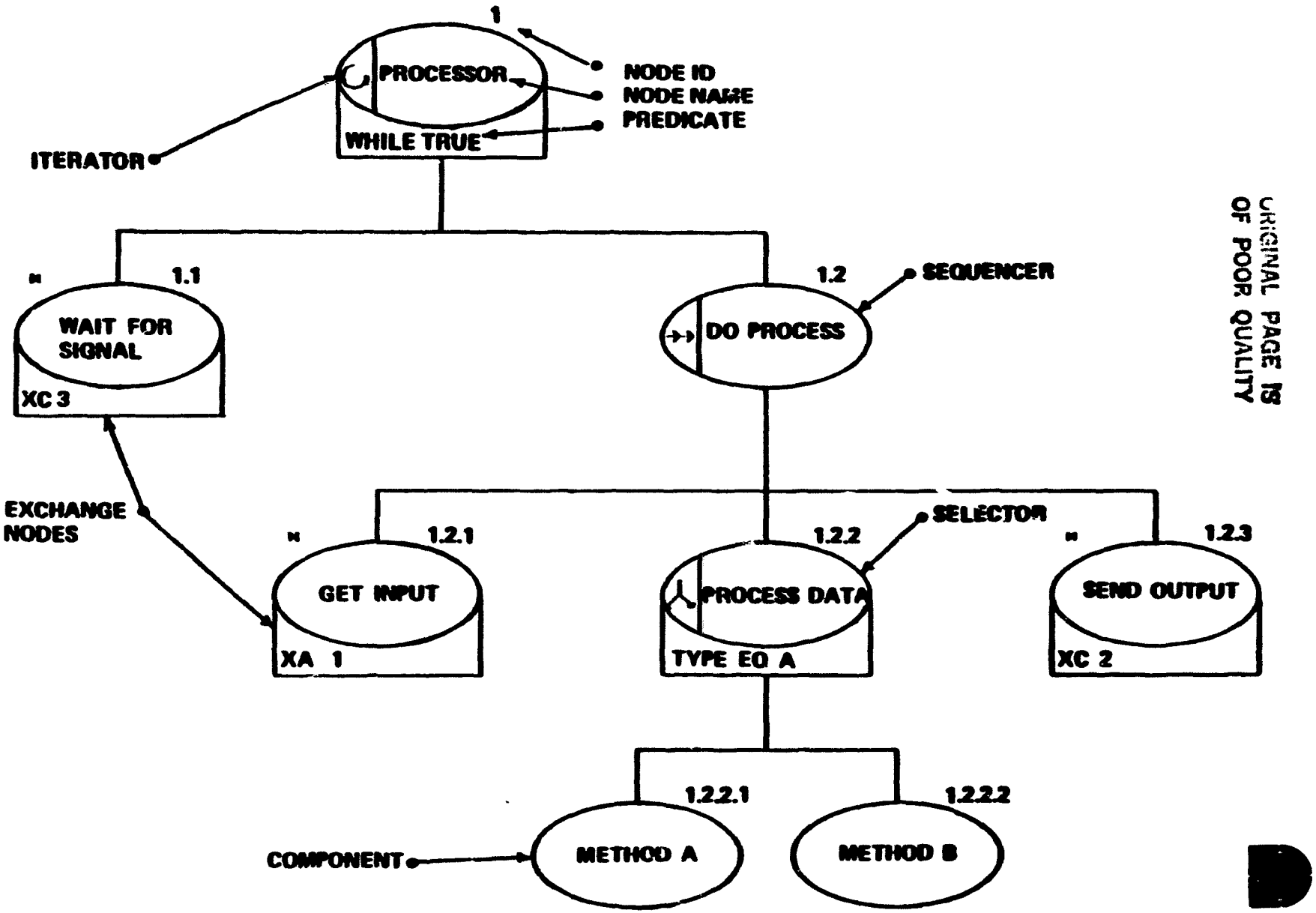
USING DARTS



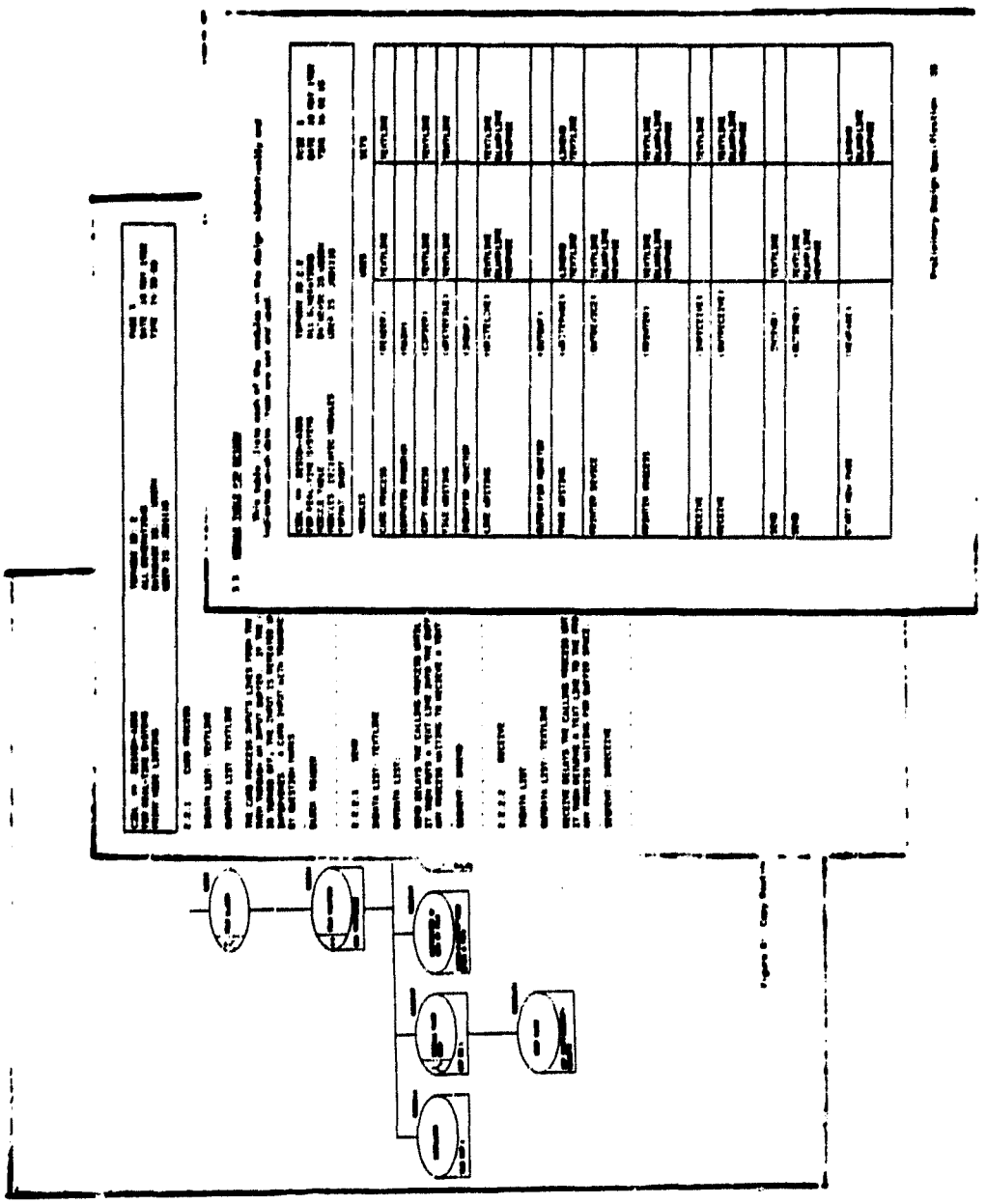
DARTS MENUS

- Primary features
  - Darts invocation
  - Simulation
  - Utilities
- Secondary features
  - Systems management
  - Tree management
  - Graphics
  - Tables
  - Analysis

PROCESS ARCHITECTURE TREE



DARTS OUTPUTS: AUTOMATED DOCUMENTATION



DARTS AUTOMATED SOFTWARE QUALITY MEASUREMENT

- Objective measure of software quality
- Uses Halstead's software science method
- Accommodates varying levels of design detail
- Automatic measurements from DARTS data base

DARTS THREE COUNTING METHODS

- **Simple**
  - **All nodes are counted the same, and all indata and outdata lists are counted**
- **Uninterpreted**
  - **Nodes are differentiated as being either functional nodes or decision nodes. Data lists are read accordingly: indata and outdata for functional nodes, and predvar lists for decision nodes. Each node is counted separately by node ID**
- **Interpreted**
  - **Nodes are counted by name and all tabs are parsed for operators and operands. Data lists are ignored**

DARTS EXAMPLE
SOFTWARE COMPONENT LEVEL

CEDL ** DESIGN AIDS
FOR REAL-TIME SYSTEMS
PLOTTER2 (11.11)
DATABASE IS TEST
OWNER IS [illegible]

PAGE 3
DATE 24 NOV 1982
TIME 13:07:51
VERSION 7.2.2.4
3 OPERATIONS

DARTS EXAMPLE
SOFTWARE COMPONENT DETAILS

ORIGINAL PAGE IS
OF POOR QUALITY

DARTS METRIC ANALYSIS

CSDL ** DESIGN-AIDS FOR REAL-TIME SYSTEMS HALSTEAD METRIC	TOPNODE ID:7.2.2.2.4 3 GENERATIONS DATABASE IS: TEST USER IS: PAS3132	PAGE 2 DATE: 24 NOV 1982 TIME: 13:12:14

COUNTING METHOD:	SIMPLE	UNINTERPRETED	INTERPRETED
DISTINCT OPERATORS	3	14	24
DISTINCT OPERANDS	20	20	21
TOTAL OPERATORS	35	31	56
TOTAL OPERANDS	55	42	49
VOCABULARY	23	34	45
DESIGN LENGTH	90	73	105
ESTIMATED LENGTH	91.2	139.7	202.3
PERCENT OFF	-1.33	-91.43	-92.65
DESIGN VOLUME	407.121	371.385	576.645
POTENTIAL VOLUME	28.529	28.529	28.529
DESIGN LEVEL	0.070	0.077	0.049
ESTIMATED DESIGN LEVEL	0.242	0.068	0.036
INTELLIGENCE CONTENT	98.696	25.264	20.594
LANGUAGE LEVEL	1.999	2.192	1.411
EFFORT	5309.715	4834.555	11655.336

DARTS SUMMARY

- User friendly
- Hierarchical structure
- Can accommodate reai-time software
- Static quality analysis
- Dynamic analysis
- Documentation support
- Design traceability

DARTS FUTURE

- **Near term**
  - **Validate existing metrics and add others to DARTS design quality analysis feature. (This effort is presently under contract to Rome Air Development Center #F30602-82-C-0130)**
- **Long term**
  - **Consider DARTS as a part of an integrated software engineering support environment**

PANEL #3

SOFTWARE ERRORS

T. Ostrand/E. Weyuker, Sperry Univac/Courant Institute
E. Solloway/W. Johnson/S. Draper, Yale/University of California
D. Buckland, Reifer Consultants

SOFTWARE ERROR DATA COLLECTION AND CATEGORIZATION

Thomas J. Ostrand
Software Technology Research
Sperry Univac
Blue Bell, PA 19424

Elaine J. Weyuker
Courant Institute
New York University
New York, NY 10012

Seventh Annual Software Engineering Workshop
Goddard Space Flight Center
December 1, 1982

A study has been made of the software errors detected during development of an interactive special-purpose editor system. This product has been followed during nine months of coding, unit testing, function testing, and system testing. Detected errors and their fixes have been described by testers and debuggers. To help analyze the relationship of error characteristics to the various aspects of the software development process, a new error categorization scheme has been developed. Within this scheme, 174 errors were classified. For each error, we asked the programmers to select the most likely cause of the error, report the stage of the software development cycle in which the error was created and first noticed, and the circumstances of its detection and isolation, including time required, techniques tried, and successful techniques.

The programmers were also asked to give a written description of the error, its symptoms, and its correction. The new error categorization scheme was developed from these descriptions. Four generic attributes or dimensions of software errors were identified; an error is classified by assigning it a value for each dimension. The four dimensions and their possible values reflect the specific errors studied for this project. As the study is extended to development projects producing different types of software and different types of errors, the dimensions and their values will be extended as needed.

The four present error dimensions are:

- Major Category - a broad description of the error, identifying the type of code that was changed to make the correction. The seven major categories into which errors from the interactive editor have been put are:

Data Definition	Code that defines constants, storage areas, control codes, transfer tables, etc.
Data Handling	Code that modifies or initializes the values of variables.
Decision	Code that evaluates a condition and branches according to the result.
Decision plus Processing	Code that evaluates a condition and performs a specific computation if the condition is satisfied.
Documentation	Written description of the product.
System	An error external to the program itself, including operating system, compiler, hardware, etc.
Not an Error	Problem reports that are resolved without changing any part of the system or product.

- Type - more specific information modifying the major category.

- Presence - whether the error involves omitted, superfluous, or incorrect code.

- Use - whether the error involves an initialization, update, or setting of data.

The interactive editor system is a small project; three programmers spent about two person-years in its development and testing. The source code consists of about 9000 lines of high-level language, and 1000 assembler instructions. Obviously, this small size and the limited number of programmers prevent us from drawing any far-reaching conclusions from the error data. We view this study as a pilot effort whose primary results have been the experience gained in collecting software error data, creation of the error categorization scheme, and the formation of a number of hypotheses about software development and validation methods.

The experience will be applied to future error studies, which are planned on other software projects. The categorization scheme will be used to classify the errors reported from these projects, and will be extended

with additional attributes and major categories. The hypotheses will be examined in light of the error information collected from these additional projects.

Even within the small scope of the data collected from the editor project, some interesting relationships were observed between an error's major category on the one hand, and the error's presence and the type of testing which detected it on the other. Among decision-related errors (major category decision or decision plus processing), 81% were omitted code and 19% were incorrect code. For data definition errors, 31% were omissions and 69% incorrect. Data handling errors were split approximately evenly between omitted and incorrect code, as was the entire set of errors reported on. Previous error studies have reported a similar majority of omitted code errors involving decisions. In five software projects monitored at TRW [13], decision-related errors of omission ranged from 65% to 96% of all decision-related errors. In turn, the decision errors were 11% to 36% of all errors. Glass [7] counted 60 "omitted logic" errors out of a total of 200.

At the present time the interactive editor has just been released to customers; all errors reported to date have been detected during internal testing activities. A very large majority of these pre-release errors were isolated and corrected quickly. Less than 1 hour per error was expended to isolate 79% of the errors and to correct 71%. Within 4 hours, 88% were isolated and 90% were corrected. These figures are similar to the effort measured by Weiss [16] and Presson [11].

Since our error collection spanned the entire development process, we were able to observe substantial differences between the effectiveness of unit and function testing for detecting some categories of errors. Unit testing is performed by the software project's original coders, testing

their own modules or procedures. The goal is to find errors affecting the functional behavior of these individual units. Function testing is performed on the complete product by a separate testing group. A test plan is developed from the user manual, and the test cases attempt to execute all potential user activities with the product. Unit testing detected twice as many (22 vs. 11) data handling errors as function testing did. Function testing was more successful on data definition errors (47 to 7), decision errors (20 to 10), and decision plus processing errors (25 to 1).

These figures may reflect an inherent weakness in the ability of unit testing to detect certain categories of errors. Another possibility, however, is that unit testing is most successful when errors occur primarily through programmer failings, and least successful when errors are due to "high-level" problems such as ambiguous or incomplete specifications. This interpretation is supported by the programmers' choices of reasons for errors occuring. The three most commonly cited error causes were programmer error (68%), poor specifications (13%), and clerical (9%). Of the 21 errors due to poor specificatons, only one was detected in unit testing, and seventeen were detected in function testing.

Errors caused by poor specifications were not only detected later than the average of all errors; they also required more effort to correct. Only 24% of specification-caused errors were fixed in under 1 hour, 52% in 1 to 4 hours, and 24% in 4 hours to 1 day and over 1 day. The relatively high correction effort for these errors illustrates the common belief that the cost of correcting an error increases when the error remains in the system during multiple phases of the development cycle. Page [10], for example, states that the correction cost approximately doubles as an error enters each successive phase. These specification-caused errors entered the system during program design, and remained undetected during coding and unit

testing. In addition, the error fixing effort reported here is only the time spent by the programmers in constructing fixes, and does not include the effort expended by an independent tester in detecting the error and supplying additional diagnostic information. If these were included, the total correction cost would be even higher.

References

[1] Amory, W. and J.A. Clapp, "A Software Error Classification Methodology", MTR-2648, Vol. VII, Mitre Corp., Bedford, MA, 30 June 1973.

[2] Baker, W.F., "Software Data Collection and Analysis: A Real-Time System Project History", RADC-TR-77-192, Rome Air Development Center, Griffis AFB, NY, June 1977.

[3] Basili, V.R., "Data Collection Validation and Analysis", Draft Software Metrics Panel Final Report, ed. A.J. Perlis, F.G. Sayward, and M. Shaw, Washington, DC, 30 June 1980.

[4] Basili, V.R. and D.M. Weiss, "Analyzing Error Data in the Software Engineering Laboratory", Fourth Minnowbrook Workshop on Software Performance Evaluation, Blue Mtn. Lake, NY, August 1981.

[5] Basili, V.R., M.V. Zelkowitz, F.E. McGarry, R.W. Reiter, W.F. Truszkowski, and D.M. Weiss, "The Software Engineering Laboratory", Tech. Report TR-535, U. Maryland Computer Science Center, College Park, MD, May 1977.

[6] Endres, A., "An Analysis of Errors and Their Causes in System Programs", IEEE Trans. Softw. Eng., Vol SE-1, June 1975, 140-149.

[7] Glass, R.L., "Persistent Software Errors", IEEE Trans. Softw. Eng., Vol. SE-7, March 1981, 162-168.

[8] Litecky, C.R. and G.B. Davis, "A Study of Errors, Error-Proneness, and Error Diagnosis in Cobol", Comm. ACM, Vol. 19, January 1976, 33-37.

[9] Mendis, K.S. and M.L. Gollis, "Categorizing and Predicting Errors in Software Programs", Proc. 2nd AIAA Computers in Aerospace Conf., Los Angeles, October 1979, 300-308.

[10] Page, J., "Evaluating the Effects of an Independent Verification and Validation Team", Proc. 6th Ann. Software Eng. Workshop, Goddard Space Flight Center, Greenbelt, MD, December 1981.

[11] Presson, P.E., "A Study of Software Errors on Large Aerospace Projects", Proc. Nat. Conf. on Software Technology and Management, Alexandria, VA, October 1981

[12] Schneidewind, N. and H. Hoffman, "An Experiment in Software Error Data Collection and Analysis", IEEE Trans. Softw. Eng., Vol SE-5, May 1979, 276-286.

[13] Thayer, T.A., M. Lipow, and E.C. Nelson, Software Reliability, TRW Series of Software Technology, Vol. 2, North-Holland, Amsterdam, 1978.

[14] Thibodeau, R., "The State-of-the-Art in Software Error Data Collection and Analysis - Final Report", General Research Corp., Huntsville, AL, Jan. 31, 1978.

[15] Weiss, D.M., "Evaluating Software Development by Error Analysis: The Data from the Architecture Research Facility", J. Systems and Software, Vol. 1, 1979, 57-70.

[16] Weiss, D.M., "Evaluating Software Development by Analysis of Change Data", Tech. Report TR-1120, U. Maryland Computer Science Center, College Park, MD, November 1981

THE VIEWGRAPH MATERIALS

for the

T. OSTRAND/E. WEYUKER PRESENTATION FOLLOW

SOFTWARE ERROR DATA COLLECTION AND CATEGORIZATION

THOMAS OSTRAND
SOFTWARE TECHNOLOGY
SPERRY UNIVAC

ELAINE WEYUKER
COURANT INSTITUTE
NEW YORK UNIVERSITY

SEVENTH SOFTWARE ENGINEERING WORKSHOP
GODDARD SPACE FLIGHT CENTER
DECEMBER 1, 1982

PROJECT DESCRIPTION

PURPOSE: IMPLEMENT A LANGUAGE-SPECIFIC INTERACTIVE EDITOR.

FEATURES:
- TEMPLATES FOR DATA DEFINITIONS AND CONTROL STRUCTURES
- FORMATTING OF SOURCE CODE.
- DYNAMIC SYNTAX CHECKING
- PROMPTING FOR REQUIRED PROGRAM SECTIONS.

SCHEDULE:

- SPECIFICATION AVAILABLE	11/80
- CODING BEGAN	4/81
- FUNCTION TESTING BEGAN	11/81
- SYSTEM TESTING BEGAN	4/82
- CUSTOMER TESTING BEGAN	6/82
- RELEASE	11/82

PROJECT DESCRIPTION

STAFF: - 1 FULL TIME, 2 PART-TIME PROGRAMMERS

SIZE: - SOURCE CODE 9000 LINES HLL
1000 LINES AL

- OBJECT CODE 70,000 BYTES

CHANGE INFORMATION COLLECTED FROM PROGRAMMERS

CHECK-OFF INFORMATION

- PROBLEM DETECTION METHODS
- PROBLEM ISOLATION METHODS
- ORIGINAL CODER
- TIME REQUIRED FOR ERROR ISOLATION AND ERROR FIXING
- SIZE OF CHANGE
- WHEN PROBLEM WAS NOTICED
- WHEN PROBLEM WAS CREATED
- WHY DID THE PROBLEM OCCUR

CHANGE INFORMATION COLLECTED FROM PROGRAMMERS

WRITTEN INFORMATION

- DATES
- NAMES OF CHANGED UNITS
- DESCRIPTIONS OF
  - PROBLEM SYMPTOMS
  - ACTUAL PROBLEM
  - FIX
- OTHER MISCELLANEOUS INFORMATION

ERROR CATEGORIZATION METHODS

AMORY & CLAPP	MITRE
ENDRES	IBM DOS SOFTWARE
THAYER ET AL	TRW APPLICATIONS
GLASS	BOEING APPLICATIONS

CHARACTERISTICS OF THESE METHODS ARE:

- TREE SCHEME FOR CATEGORIZATION
- AMBIGUOUS, OVERLAPPING, INCOMPLETE CATEGORIES
- TOO MANY CATEGORIES
- FAILURE TO DISTINGUISH BETWEEN:
    - SYMPTOMS OF AN ERROR
    - DESCRIPTIVE CHARACTERISTICS OF AN ERROR
    - CAUSE OF AN ERROR'S EXISTENCE

ATTRIBUTES IN OUR CURRENT SCHEME

- MAJOR CATEGORY
- TYPE
- PRESENCE
- USE

ATTRIBUTES

MAJOR CATEGORY

DATA DEFINITION	- DEFINE CONSTANTS, STORAGE AREAS, CONTROL CODES, ETC.
DATA HANDLING	- SET, INITIALIZE, OR MODIFY VALUES OF VARIABLES.
DECISION	- EVALUATE A CONDITION AND BRANCH ACCORDING TO THE RESULT.
DECISION & PROCESS	- EVALUATE A CONDITION AND PERFORM A COMPUTATION.
DOCUMENTATION	- DESCRIPTION OF PRODUCT OR CODE
CLERICAL	- TYPING, HANDWRITING
SYSTEM	- PROBLEM IN THE ENVIRONMENT EXTERNAL TO THE PROGRAM AND ITS DOCUMENTATION.
NOT AN ERROR	- PROBLEM RESOLVED WITHOUT CHANGING THE PRODUCT OR SYSTEM

ATTRIBUTES

TYPE: MODIFIES THE MAJOR CATEGORY

- FOR ERRORS INVOLVING DATA:

ADDRESS - IDENTIFIES LOCATION IN MEMORY.

EXAMPLES: ARRAY INDEX, LIST POINTER, TABLE NAME, OFFSET INTO A DEFINED STORAGE AREA.

CONTROL - DETERMINES APPROPRIATE FLOW OF CONTROL

DATA - PRIMARY INFORMATION WHICH IS READ, WRITTEN, OR PROCESSED.

- FOR ERRORS INVOLVING DECISIONS:

LOOP

MULTIPLE-WAY BRANCH

ATTRIBUTES

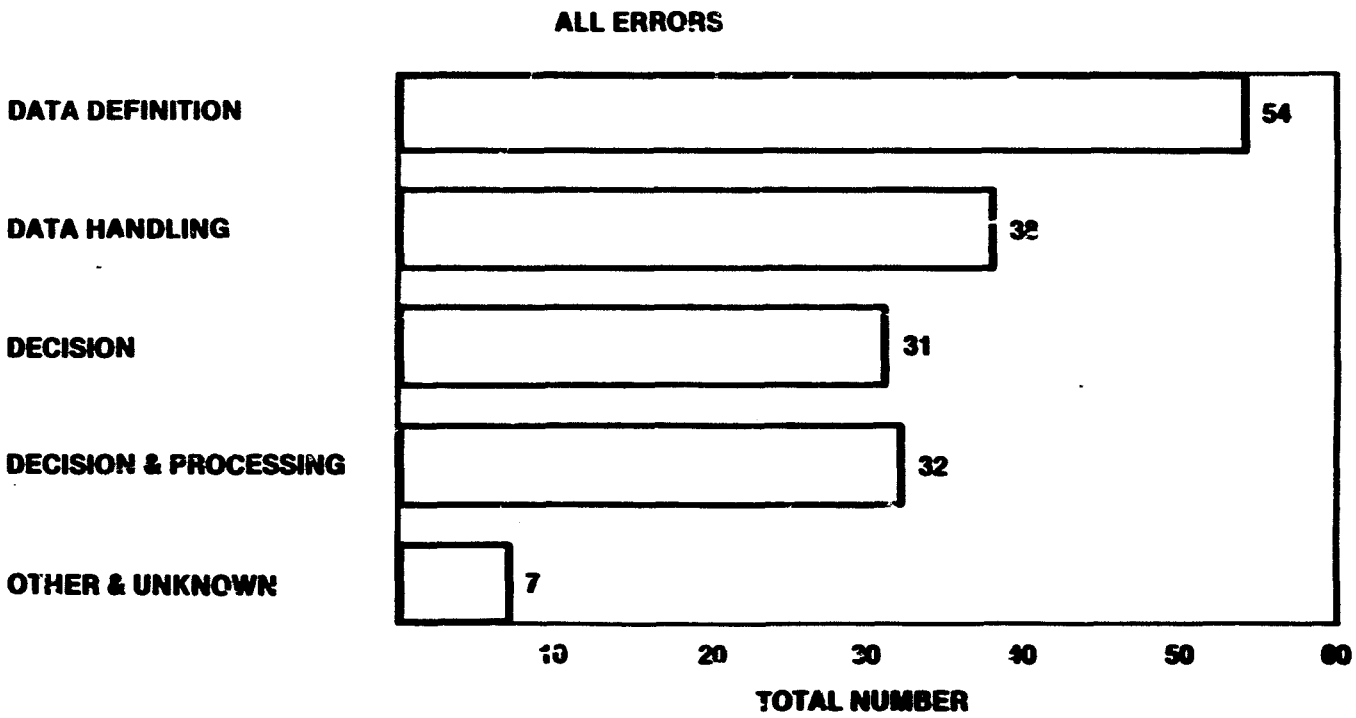
PRESENCE: CODE WAS

- OMITTED - LEFT OUT
- SUPERFLUOUS - PRESENT, BUT NOT NEEDED
- INCORRECT - PRESENT, AND HAD TO BE CHANGED.

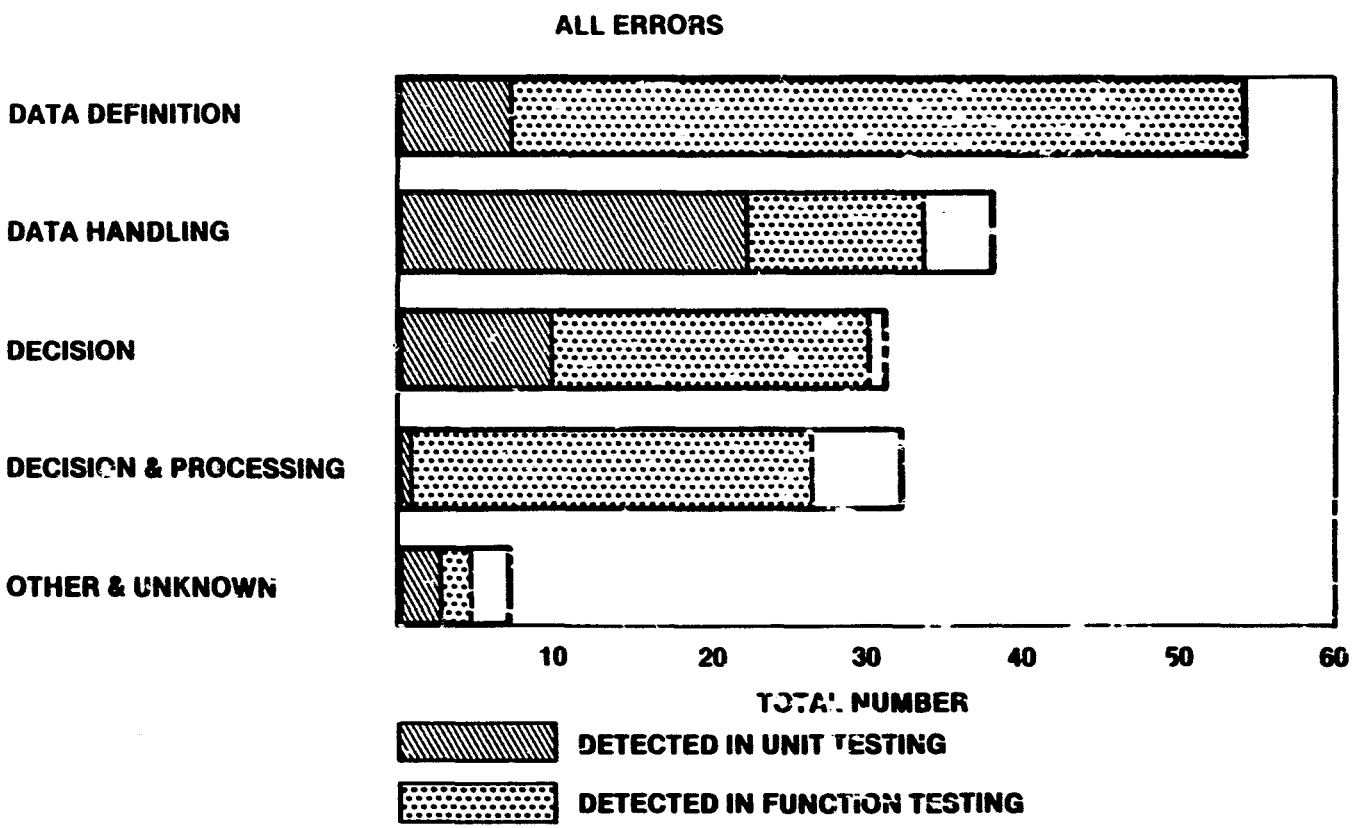
USE: THE TYPE OF OPERATION PERFORMED ON DATA

- SET
- INITIALIZE
- UPDATE

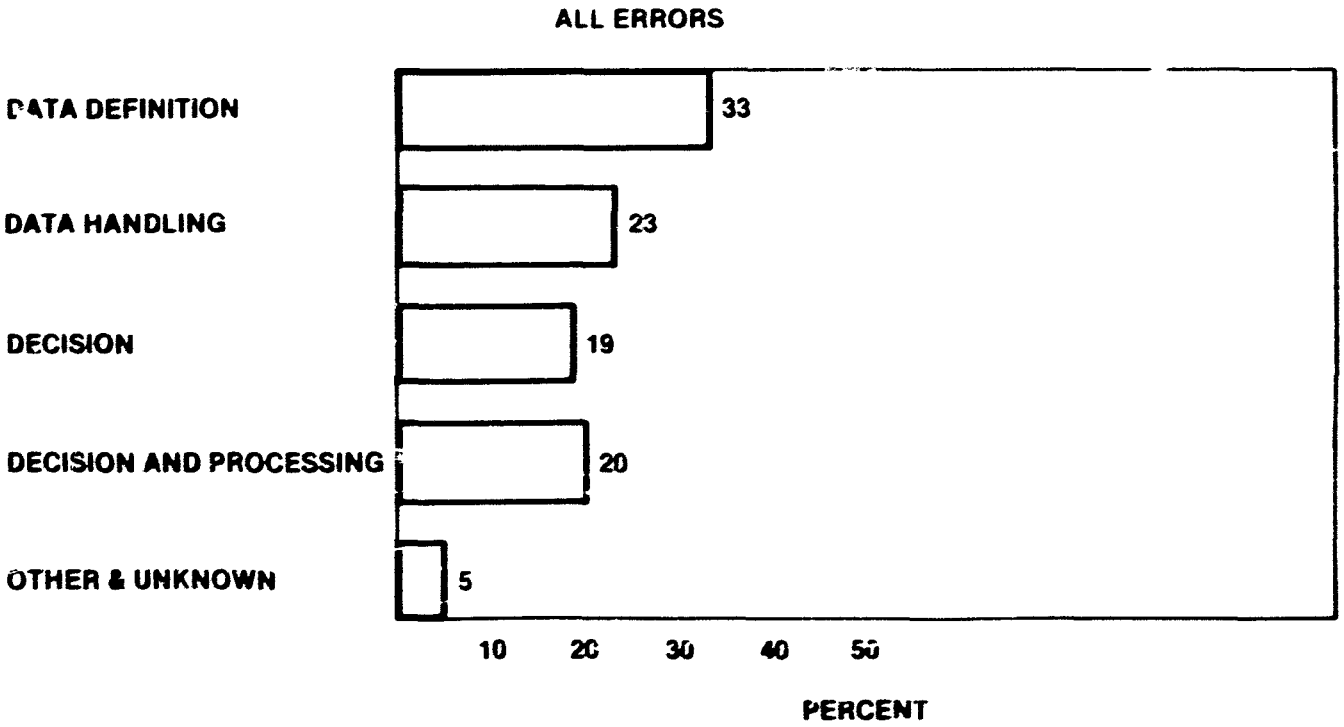
Major Categories of Non-Clerical Errors



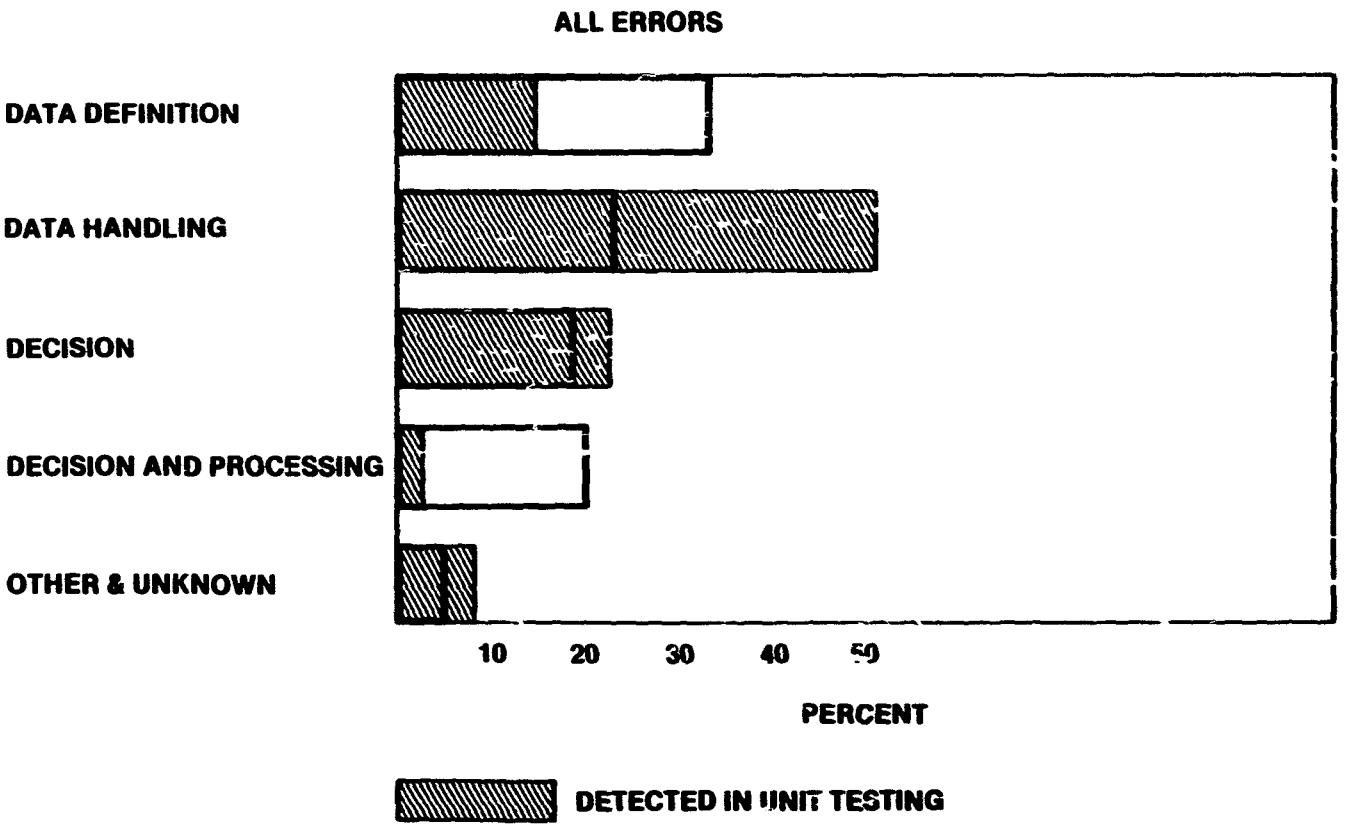
Major Categories of Non-Clerical Errors



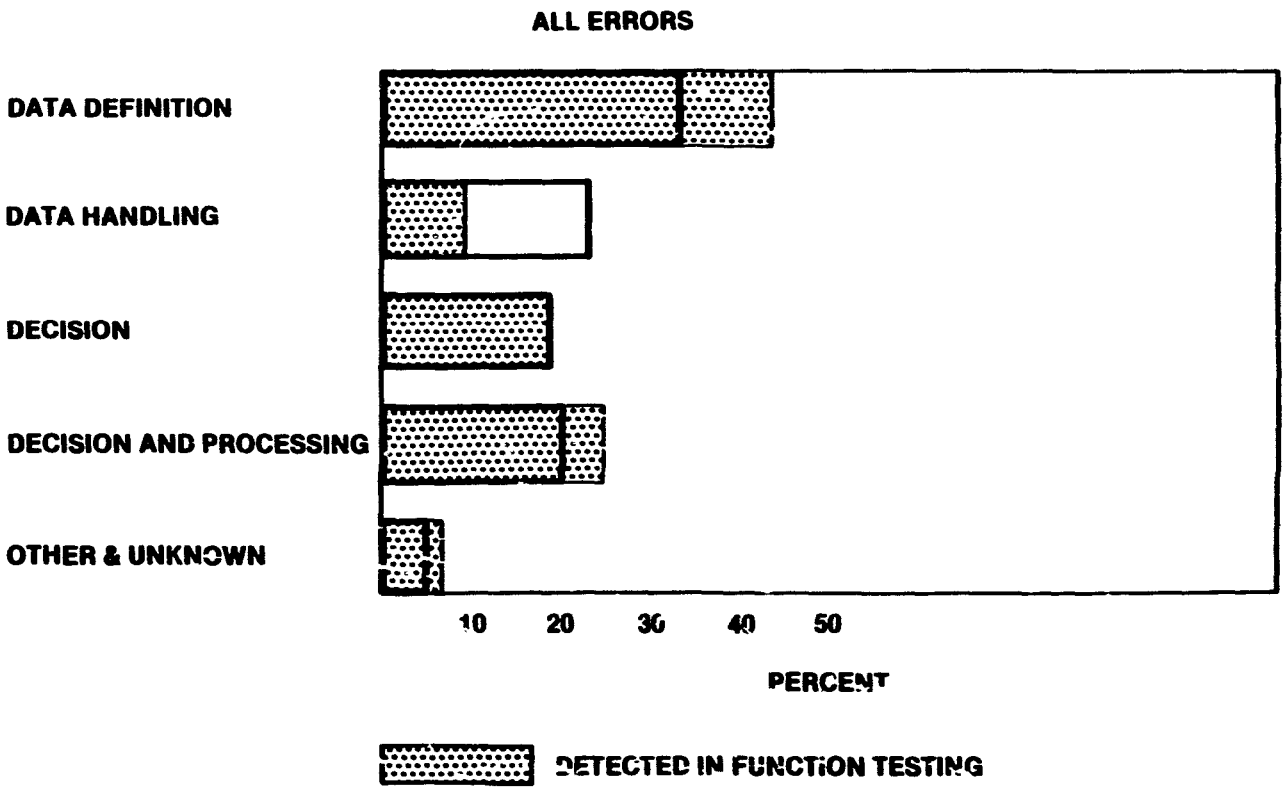
Major Categories of Non-Clerical Errors



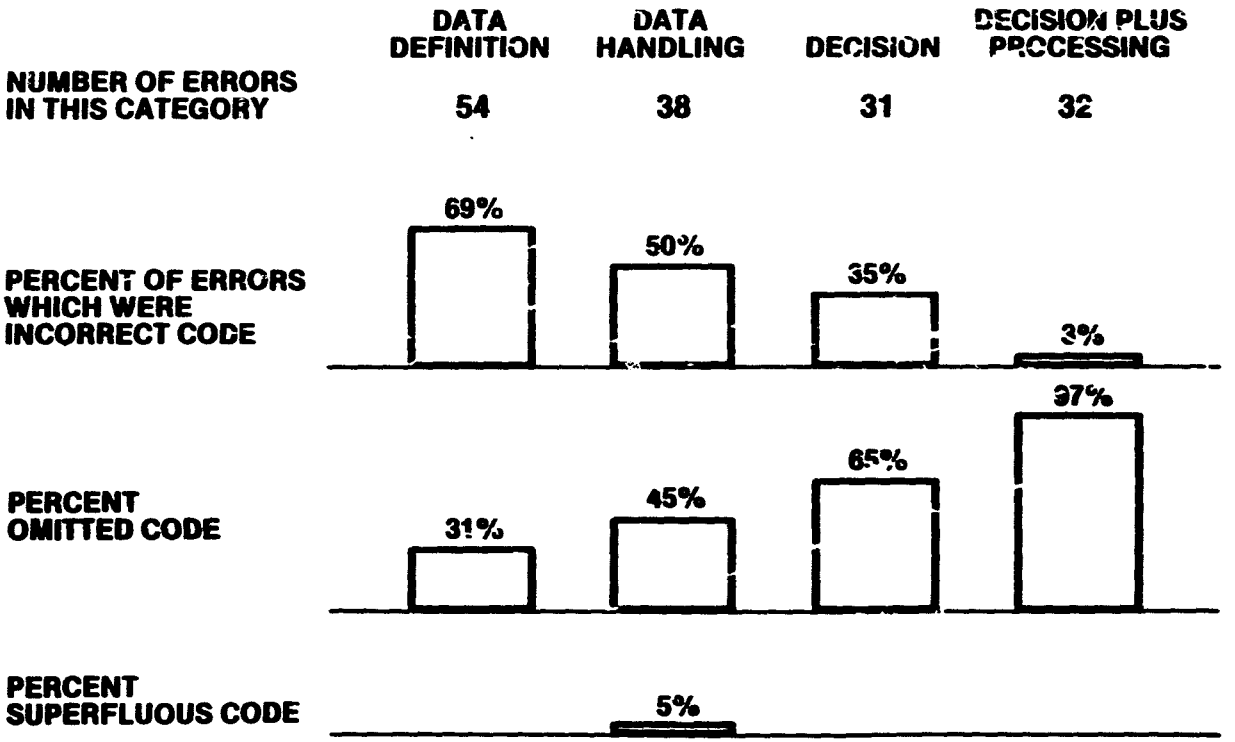
Major Categories of Non-Clerical Errors



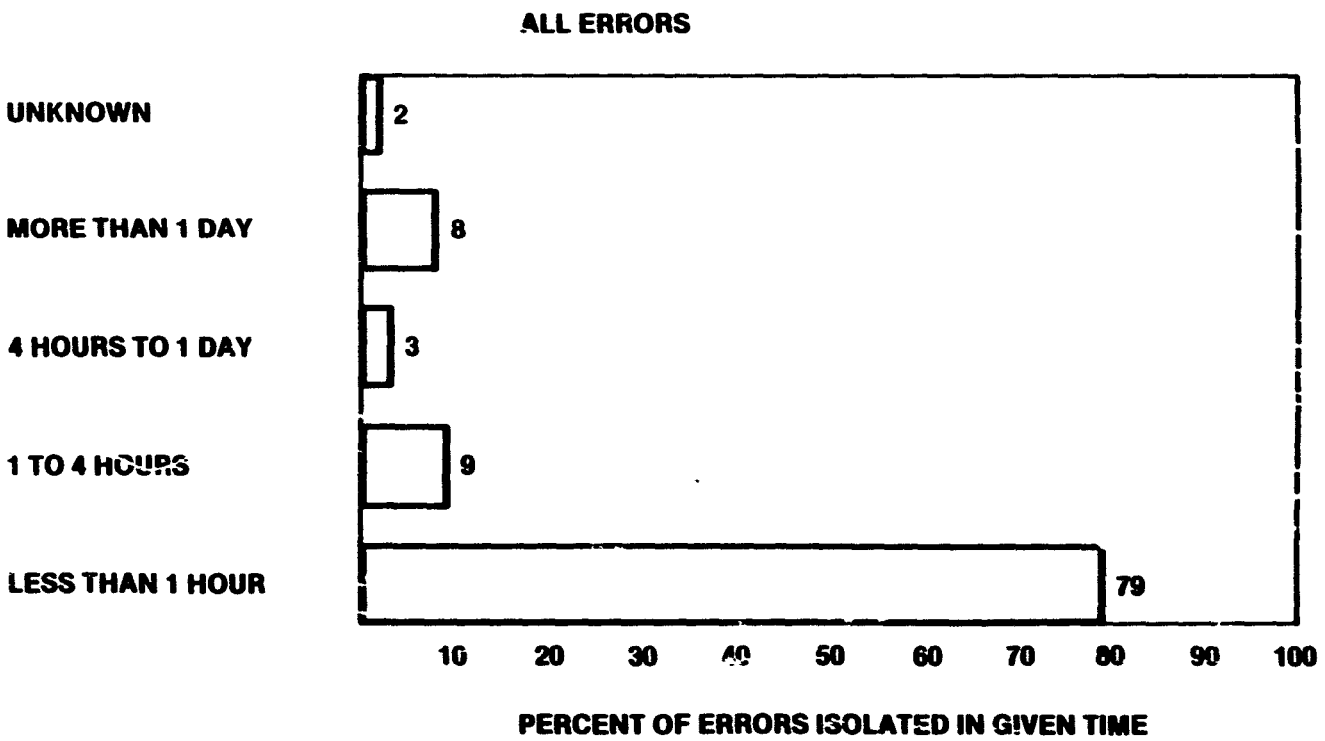
Major Categories of Non-Clerical Errors



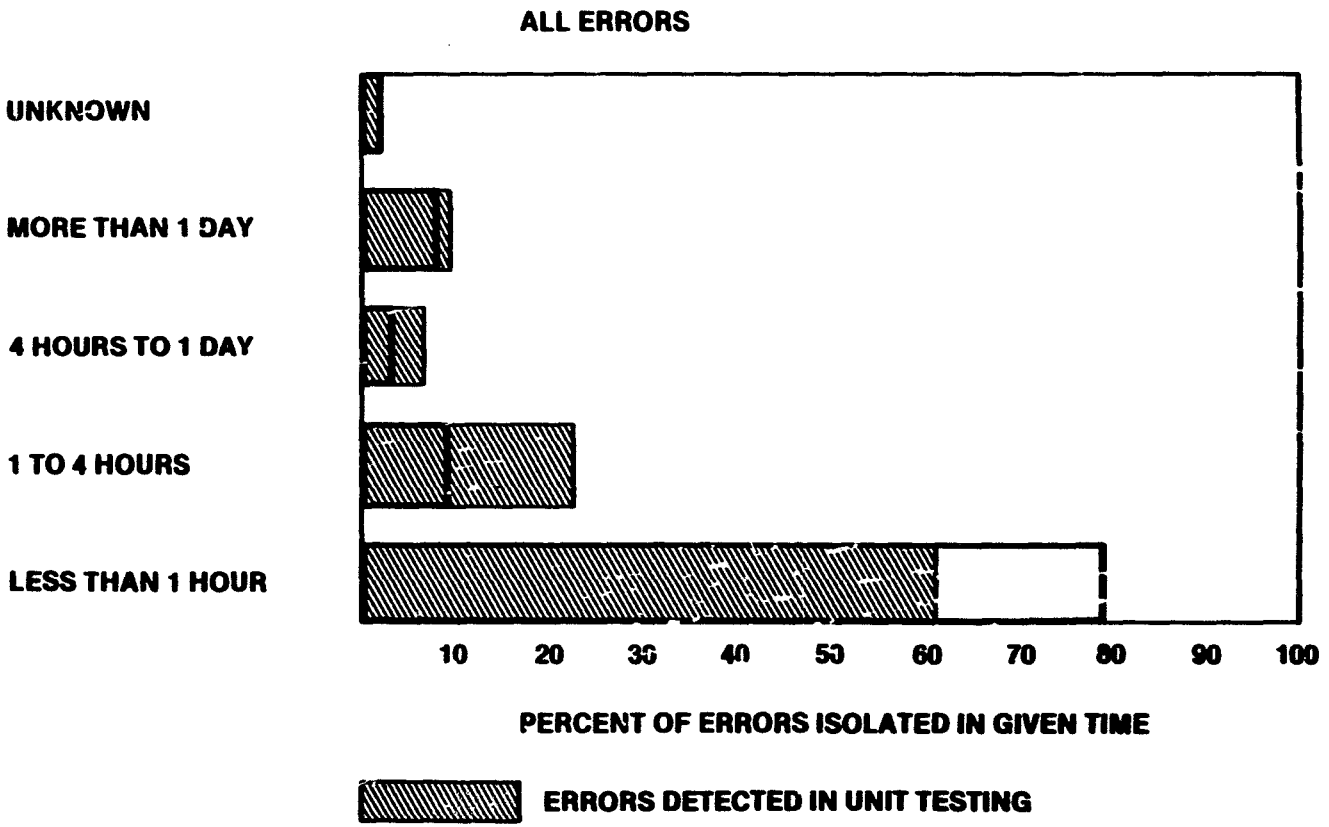
Error Presence Attribute for Each Major Category



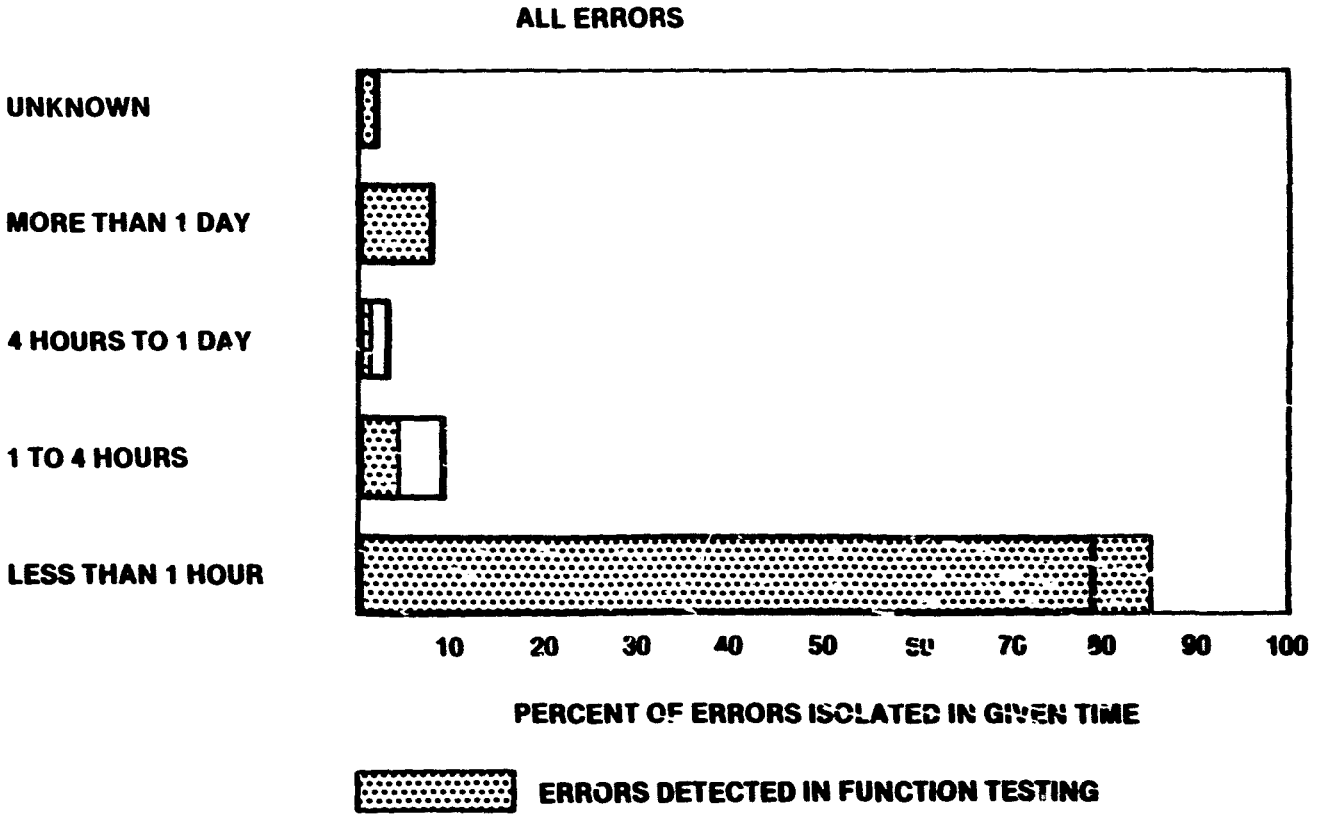
Error Isolation Effort



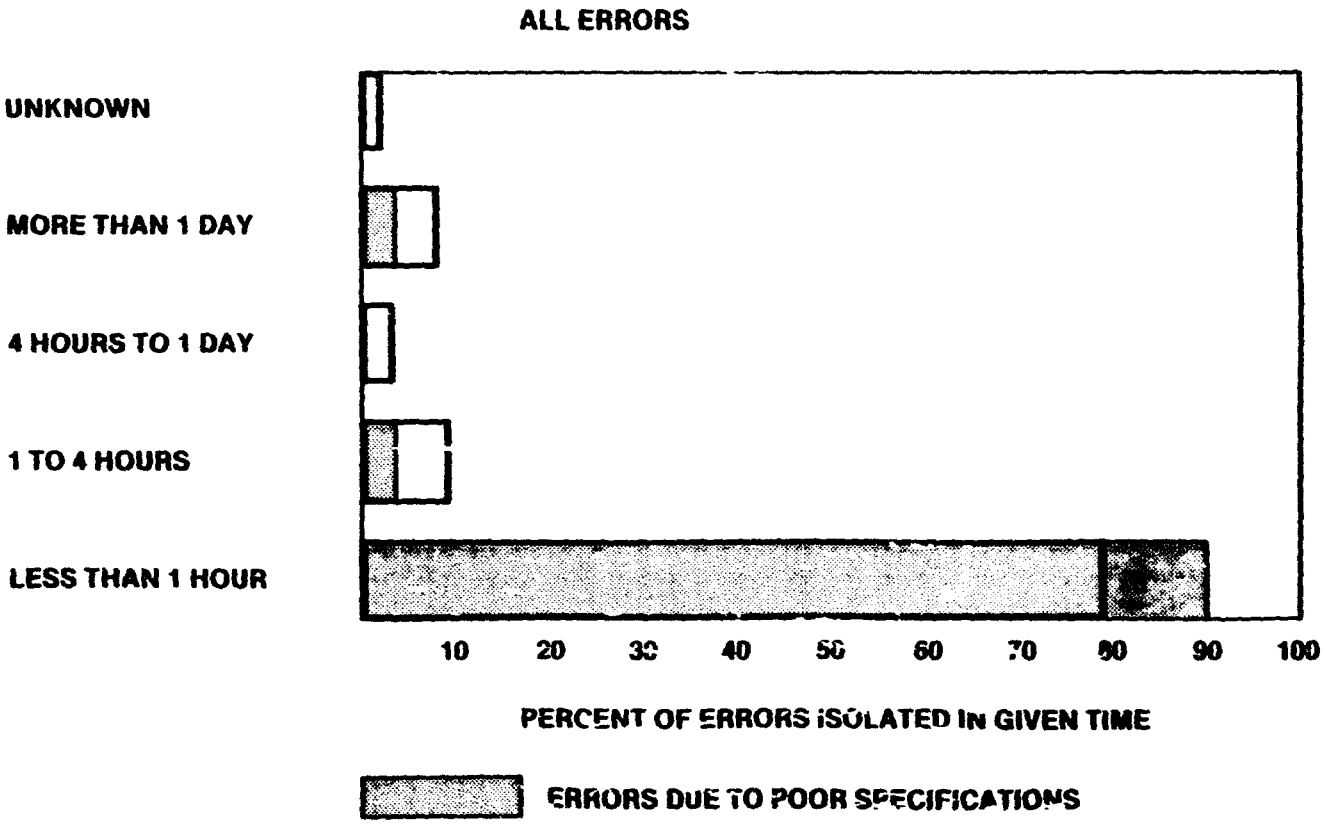
Error Isolation Effort



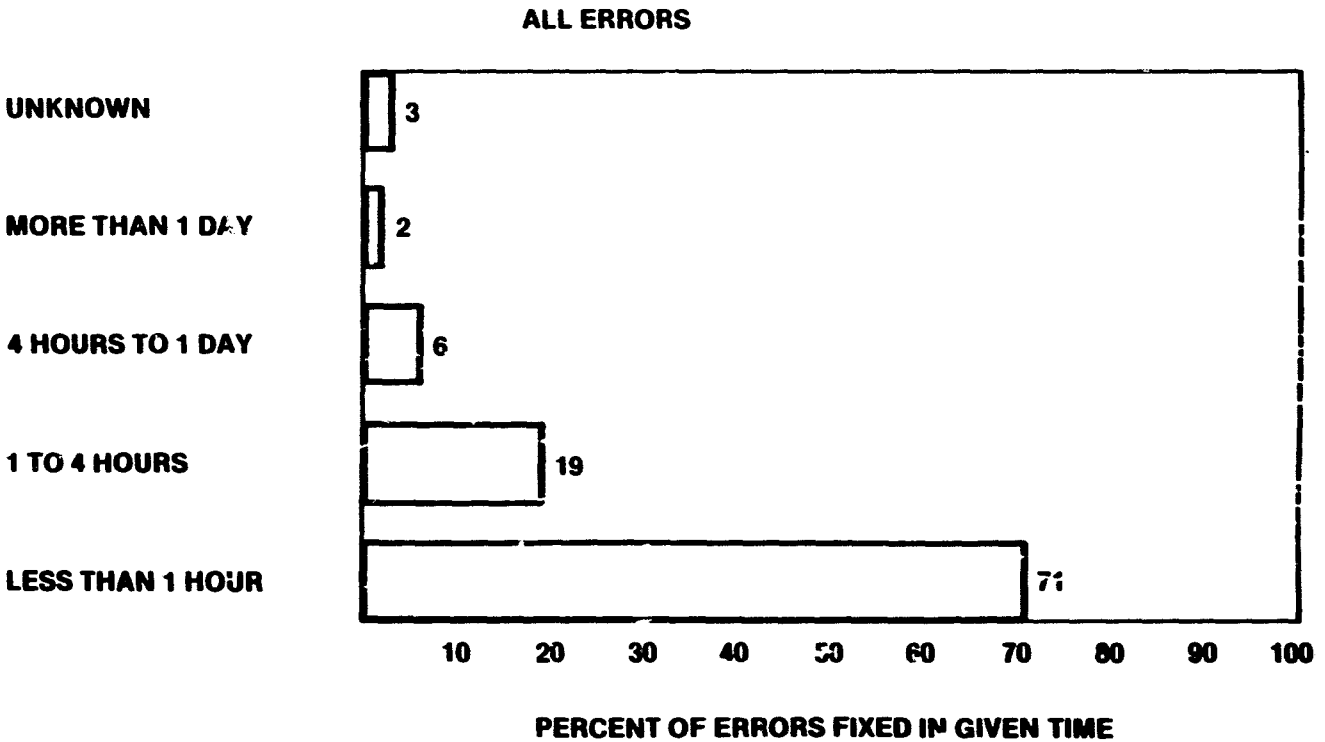
Error Isolation Effort



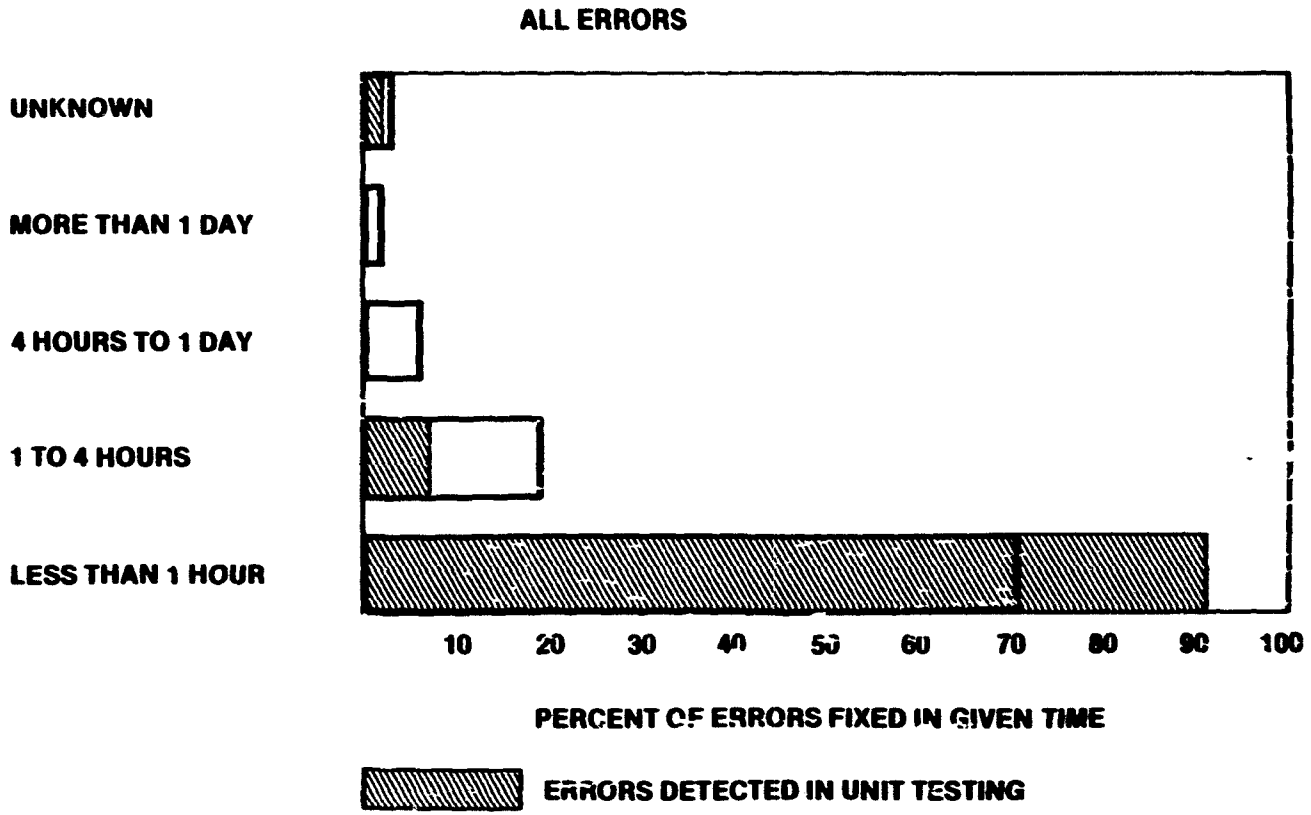
Error Isolation Effort



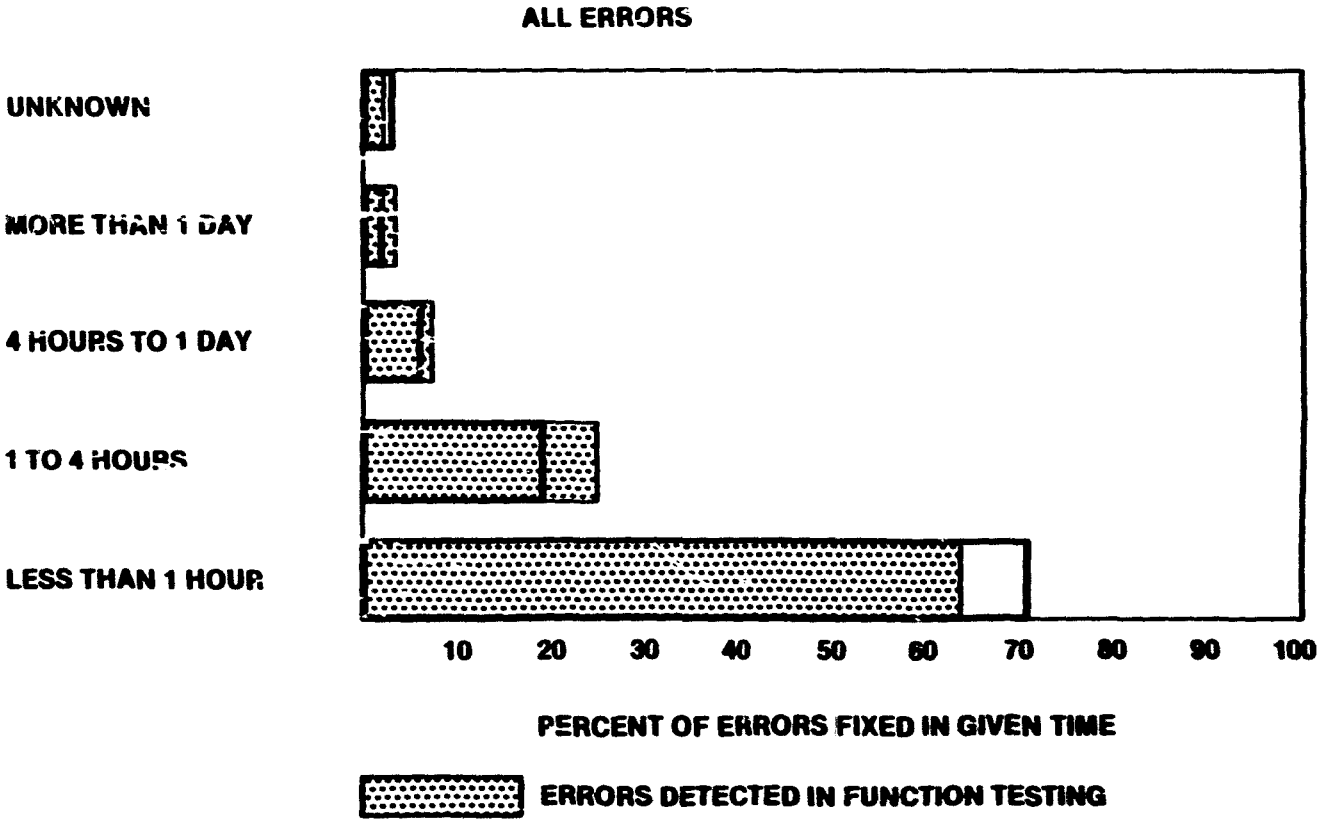
Error Fixing Effort



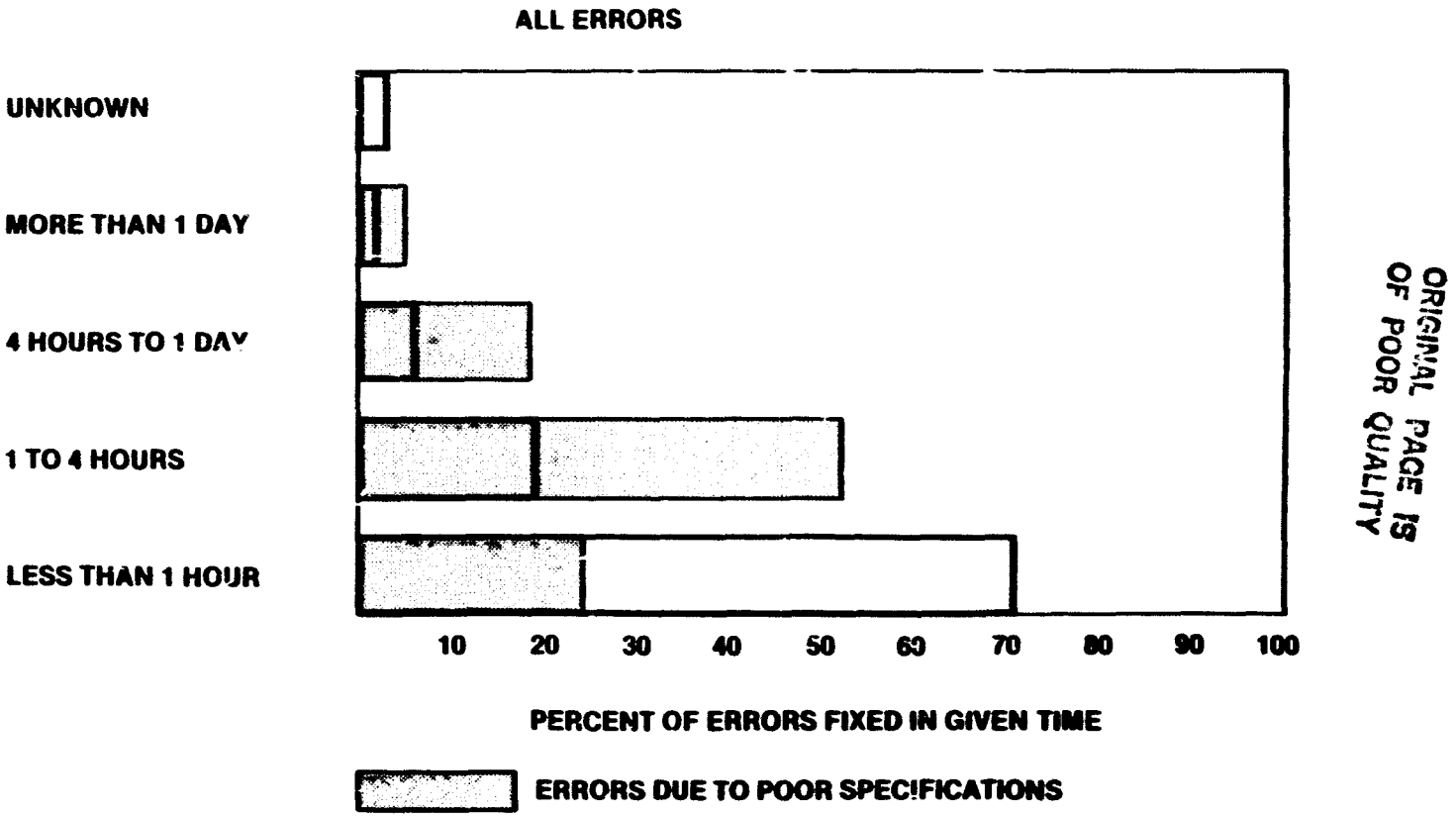
Error Fixing Effort



Error Fixing Effort



Error Fixing Effort



SUMMARY OF RESULTS

- UNIT TESTING DETECTS DATA HANDLING ERRORS WELL.

- FUNCTION TESTING DETECTS DECISION-RELATED ERRORS AND DATA DEFINITION ERRORS WELL.

- LARGE MAJORITY OF DECISION-RELATED ERRORS ARE OMISSIONS. (AGREES WITH PRIOR STUDIES).

- MOST ERRORS DETECTED BEFORE RELEASE ARE ISOLATED AND CORRECTED WITH LITTLE EFFORT. (AGREES WITH WEISS AND PRESSON).

- SPECIFICATION-CAUSED ERRORS ARE MORE DIFFICULT TO CORRECT THAN OTHERS.

CONCLUSIONS OR HYPOTHESES

- MULTI-DIMENSIONAL ERROR CATEGORIZATION SCHEME IS EASIER TO USE AND MORE USEFUL FOR APPLICATIONS THAN TRADITIONAL TREE SCHEMES.

- CODE COVERAGE IS UNSATISFACTORY AS A BASIS FOR TEST CASE GENERATION AND AS A MEANS OF ASSESSING TEST ADEQUACY, BECAUSE OF THE LARGE NUMBER OF ERRORS INVOLVING OMITTED CODE.

- UNIT TESTING IS AN INHERENTLY WEAK METHOD FOR DETECTION OF ERRORS CAUSED BY POOR SPECIFICATIONS.

- EFFORT SPENT IN PRODUCING HIGH-QUALITY SPECIFICATIONS WILL SUBSTANTIALLY REDUCE THE COST OF CORRECTING SOFTWARE.

Classifying Bugs is a Tricky Business[1]

W. Lewis Johnson *
Stephen Draper **
Elliot Soloway *

* Department of Computer Science
Yale University
P.O. Box 2158
New Haven, Connecticut 06520

** Institute for Cognitive Science
University of California, San Diego Mail Code C015
La Jolla, California

1. Context: Motivation and Goals[1]

About 2 years ago we decided to build a computer-based programming tutor to help students learn to program in Pascal; we wanted the system to identify the *non-syntactic* bugs in a student's program and tutor the student with respect to the misconceptions that might have given rise to the bugs. The emphasis was on the system understanding what the student did and did not understand; we felt that simply telling the student that there was a bug in line 14 was not sufficient --- since oftentimes the bug in line 14 was really caused by a whole series of conceptual errors that could not be localized to a specific line in the program. However, in order to design the system we needed to know what bugs students did make in their programs and what misconceptions they typically labored under. On the basis of bug types found in a number of pencil-and-paper studies with student programmers (novices, intermediates, and advanced) [9, 10], we built and classroom tested a first version of such a programming tutor [11]. In the process of testing that system we instrumented the operating system on a CYBER 175 to automatically collect a copy of each syntactically correct program the student programmers attempted to execute while sitting at the terminal; we call this form of data "on-line protocols". We collected such protocols on 204 students for an entire semester (7 programming assignments). We have systematically analyzed only a small portion of these data: the basis for this paper is the hand analysis of the first syntactically correct program that students generated for their first looping assignment,[2] i.e., 204 programs.

---

[1]This work was co-sponsored by the Personnel and Training Research Groups, Psychological Sciences Division, Office of Naval Research and the Army Research Institute for the Behavioral and Social Sciences, Contract No. N00014-82-K-0714, Contract Authority Identification Number, Nr 154-492. Approved for public release; distribution unlimited. Reproduction in whole or part is permitted for any purpose of the United States Government.

[2]This problem is given in Figure 8, which will be discussed in section 4.

The story we tell in this paper deals with our experiences in analyzing these 204 on-line protocols. In particular, we will describe the observations we made in trying to build a bug classification scheme; the actual details of what bugs we found, their frequency, etc. can be found in [5]. The key observation is the following: while one might think that building a classification scheme for the bugs would be straightforward, it turns out not to be so simple; in fact, we will argue that:

> *Bugs cannot be uniquely described on the basis of features of the buggy program alone; one must also take the programmer's intentions and knowledge state into account.*

2. A Simplified Example

Consider the problem statement in Figure 1, which is a simplified version of the first looping problem that the students in our study had to solve in Pascal. From a novice's perspective the difficult part of this problem is making sure that the negative inputs are filtered out before they are processed. There are two common approaches to solving this type of problem in an Algol-like language such as Pascal. In Figure 2 we depict a solution in which a negative input causes execution of one branch of a conditional, while a non-negative input causes execution of the major computation of the loop. We call this type of structure a ***Skip-guard Plan***:[3] a conditional statement is used to guard the main computation from illegal values. Notice that one pass through the loop will be made for each input value. The second approach is given in Figure 3; here an embedded loop filters out the illegal values. Notice that one pass through the outside loop will be made for each --- and only each --- **legal** value. We call the nested loop structure an ***Embedded Filter Loop Plan.***

Write a program that reads in integers, that represent the daily rainfall in the New Haven area, and computes the average daily rainfall for the input values. If the input is a negative number, do not count this value in the average, and prompt the user to input another, legal value. Stop reading when 99999 is input; this is a sentinel value and should not be used in the average calculation.

Figure 1: Simplified Looping Problem

Now consider the buggy program in Figure 4. The problem with this program is that if the user first types a negative input, and then types the sentinel value 99999, this value will --- incorrectly --- be processed as a legitimate value. A number of questions come to mind:

1. How should we classify this bug?
2. What piece of code is to blame?
3. What mental error on the student's part might have caused this bug?

---

[3]See [8, 3, 9]for a more complete discussion of programming plans.

```
....
....
READ(RAINFALL)
WHILE RAINFALL <> 99999 DO
    BEGIN
       IF RAINFALL < 0
          THEN
           WRITELN('BAD INPUT, TRY AGAIN')
          ELSE
           BEGIN
             TOTAL := TOTAL + RAINFALL;
             DAYS  := DAYS  + 1;
           END;
       READ(RAINFALL);

    END;
....
....
```

Figure 2: Using a ***Skip-Guard Plan***

```
....
....
READ(RAINFALL)
WHILE RAINFALL <> 99999 DO
    BEGIN
       WHILE RAINFALL < 0 DO
           BEGIN
              WRITELN('BAD INPUT, TRY AGAIN');
              READ(RAINFALL)
           END;
       IF RAINFALL <> 99999 THEN
           BEGIN
              TOTAL := TOTAL + RAINFALL;
              DAYS  := DAYS  + 1;
              READ(RAINFALL)
           END;
    END;
....
....
```

Figure 3: Using an ***Embedded Filter Loop Plan***

4. What piece of code should we change to make the program correct?

In order to answer these questions, however, we need to answer another one first:

> What programming approach was the user trying to implement? That is, did the student intend to implement the ***skip-guard plan*** or did he try to implement the ***embedded filter loop plan***?

Answers to the first 4 questions will be different depending on how we answer this last question.

We will continue this example by presenting first an argument that supports the choice of the ***skip-guard plan***, and then an argument that supports the choice of the ***embedded filter***

```
....
....
READ(RAINFALL)
WHILE RAINFALL <> 99999 DO
    BEGIN
       WHILE RAINFALL < 0 DO
           BEGIN
              WRITELN('BAD INPUT, TRY AGAIN');
              READ(RAINFALL)
           END;
       TOTAL := TOTAL + RAINFALL;
       DAYS  := DAYS  + 1;
       READ(RAINFALL)
    END;
....
....
```

Figure 4: Sample Buggy Program

loop plan; we will then describe a basis for making a choice between the two competing positions. Consider, then, Figure 5 in which we depict the buggy program again, plus a generalized, template version of the ***skip-guard plan***. We can describe the buggy program in terms of a difference description between it and the generalized plan. As shown in Figure 5, there are 3 differences:

1. need an IF instead of a WHILE inside the loop,
2. have an extra read inside the loop,
3. will always execute the processing steps since there is no way to skip around the processing.

The first difference is a plausible bug for a novice to make; in our examination of novice programs we have seen novices confuse IF and WHILE: students sometimes construct a loop with simply an IF, and sometimes they use just the test part of the WHILE statement[4] [2, 6]. Similarly, the second difference is also plausible for novices; again, we have found that novices often add bits of spurious code, oftentimes attempting to mimic the redundancy they often use in formulating plans and actions in the real world. Finally, if we assume that the programmer really meant to simply test RAINFALL, then all that is missing is an ELSE to cause the skip around the computation; novices notoriously have trouble with the ELSE parts of conditionals. Thus, the buggy code in Figure 5 is not that different from the ***skip-guard plan***; ***when considering differences from only this plan*** it is entirely conceivable that the novice programmer was trying to implement this plan in his code.

---

[4]While this may seem strange to us as expert programmers, if we take a moment to reflect, we can see that using WHILE for a conditional and a loop, and IF for only the conditional part is somewhat arbitrary, given their meanings in English.

```
....
....
READ(RAINFALL)
WHILE RAINFALL <> 99999 DO
    BEGIN
       WHILE RAINFALL < 0 DO
           BEGIN
              WRITELN('BAD INPUT, TRY AGAIN');
              READ(RAINFALL)
           END;
       TOTAL := TOTAL + RAINFALL;
       DAYS  := DAYS  + 1;
       READ(RAINFALL)
    END;
....
....
```

Skip-Guard Plan

```
IF x < min
   THEN
      BEGIN
        print error message
      END
   ELSE
      BEGIN
        process input
      END
```

BUG DESCRIPTION:

1. need an IF instead of a WHILE
2. have an extra READ in inner loop
3. missing ELSE; processing of input will never be skipped

Figure 5: Bug Description Assuming ***Skip-Guard Plan***

Now consider Figure 6 in which we again depict the buggy program. This time, however, we show differences between it and a generalized, template version of an ***embedded filter loop plan***. Notice that the code matches the plan well; the only bug is a missing guard before the code that processes the input: the running total update and the counter update must be protected from including a sentinel value in the computation.

The analysis in Figures 5 and 6 would lead to different answers to the first 4 questions above. For example, if we believe that the analysis in Figure 5 is correct, we might say the following to the student:[5]

> It seems that you are having some trouble with conditional statements. For example, did you realize that there exists a statement called IF that allows you to test ....
>
> To correct your program, you might want to add an ELSE clause...

Moreover, we would classify the bugs as an (1) incorrect statement type, (2) spurious read, (3) missing ELSE. On the other hand, if we believe that the analysis in Figure 6 is correct, then we

[5]We do not want to argue about the best pedagogical strategy for interacting with the student; that in itself is a very difficult question. The particular response shown is simply meant to illustrate one type of response to this situation.

```
....
....
READ(RAINFALL)
WHILE RAINFALL <> 99999 DO
    BEGIN
       WHILE RAINFALL < 0 DO
           BEGIN
              WRITELN('BAD INPUT, TRY AGAIN');
              READ(RAINFALL)
           END;
       TOTAL := TOTAL + RAINFALL;
       DAYS  := DAYS  + 1;
       READ(RAINFALL)
    END;
....
....
```

Embedded Filter Loop Plan

```
WHILE x < min DO
    BEGIN
       print error message
       READ x
    END
sentinel guard plan
    process input
```

BUG DESCRIPTION:

1. missing conditional (guard) on processing the input

Figure 6: Bug Description Assuming ***Embedded Filter Loop Plan***

might say something like the following to the student:

> You should notice if the sentinel value follows the input of a negative value that your program will compute an incorrect average. ....

The bug type then might be a missing guard (conditional) plan.

By this time the reader's intuition is surely saying that the correct analysis of the buggy program in Figure 4 is that the programmer intended to implement an ***embedded filter loop plan***. The bug counts (3 for the ***skip-guard plan*** and 1 for the ***embedded filter loop plan***) provide quantitative *support* for this decision. However, we feel that the key in the decision process --- and the basis for our intuition --- is our *understanding* of the student's program provided by the plan analysis in Figure 5: thus, the bug categorization and bug count *follow* from our understanding of the program --- and not the other way around. We purposely choose an example over which there would be little controversy. However, the point was (1) to show how much reasoning we often do about programs implicitly, and (2) to show how different bug categorization and bug counts could be as a function of choice of intended underlying plan.

While the above decision was relatively clear, let us perturb the buggy code a bit further and see how murky these type of decisions can --- and do --- become. In Figure 7 we show three buggy program fragments; let us compare the bug categorization and bug counts using the two

alternative plans for each of the programs.

- Figure 7a
  - Using the ***embedded filter loop plan*** we get the following bug differences:
    1. the WHILE and IF keywords have been interchanged
    2. there is a missing read for a new value
    3. there is a missing guard on the subsequent input processing
  - Using the ***skip-guard plan*** we get the following bug differences:
    1. missing ELSE on the internal IF
- Figure 7b
  - Using the ***embedded filter loop plan*** we get the following bug differences:
    1. the WHILE and IF keywords have been interchanged
    2. there is a missing guard on the subsequent input processing
  - Using the ***skip-guard plan*** we get the following bug differences:
    1. spurious READ
    2. missing ELSE on the internal IF
- Figure 7c
  - Using the ***embedded filter loop plan*** we get the following bug differences:
    1. missing read for a new value
    2. there is a missing guard on the subsequent input processing
  - Using the ***skip-guard plan*** we get the following bug differences:
    1. the WHILE and IF keywords have been interchanged
    2. missing ELSE on the internal IF

We would argue that the programmer of the code in Figure 7a intended to encode a ***skip-guard plan***: again, the bug counts (3 for the ***embedded filter loop plan*** and 1 for the ***skip-guard plan***) support the intuition that it is more plausible that the programmer simply left out an ELSE, as opposed to swapping keywords, etc. However, the code in Figures 7b and c are not so easily analyzed: the bug counts are the same and the plausibility of the bug types are reasonably similar. In order to make a reasoned decision we need to bring *other* evidence from the program to bear. For example, in Figure 7b the programmer used a WHILE loop to correctly implement the outer loop; this is some evidence that he understands how and when to use this construct. Thus, we might be confident that the programmer really meant IF in the program in Figure 7b. On the other hand, the inclusion of the spurious READ is unsettling. However, the program in Figure 7c is certainly the most problematic: the bug counts are the same, the plausibility of the bugs are similar, and the additional outside information is equivocal. The moral of this program is that it can be exceedingly difficult to make decisions about plans --- and bugs --- by *simply looking at the code.*

The point of these latter examples is to illustrate how quickly the decision about what the

a

```
READ(RAINFALL)
WHILE RAINFALL <> 99999 DO
    BEGIN
       IF RAINFALL < 0 THEN
              WRITELN('BAD INPUT, TRY AGAIN')
       TOTAL  = TOTAL + RAINFALL.
       DAYS   = DAYS  + 1
       READ(RAINFALL)
    END
```

b

```
READ(RAINFALL)
WHILE RAINFALL <> 99999 DO
    BEGIN
       IF RAINFALL < 0 THEN
           BEGIN
              WRITELN('BAD INPUT, TRY AGAIN'),
              READ(RAINFALL).
           END,
       TOTAL  = TOTAL + RAINFALL,
       DAYS   = DAYS  + 1,
       READ(RAINFALL)
    END.
```

c

```
READ(RAINFALL)
WHILE RAINFALL <> 99999 DO
    BEGIN
       WHILE RAINFALL < 0 DO
              WRITELN('BAD INPUT, TRY AGAIN'),
       TOTAL  = TOTAL + RAINFALL,
       DAYS   = DAYS  + 1,
       READ(RAINFALL)
    END
```

Figure 7: Clouding the Waters: Additional Buggy Programs

programmer intended gets murky, and how additional information outside the buggy area needs to be brought to bear. We see again that for the programs in Figure 7 the bug categorization and bug frequencies change depending on what decision is made about the programmer's intention.

Finally, the fact that the programs we have shown are *novices'* programs is really irrelevant to the point in question: the problem is that the intention of the programmer effects the bug categorization and the bug count. Quite reasonably, we would not expect a professional programmer to mistake an IF for a WHILE. The observation that we would not expect this particular confusion would in fact aid us in inferring the intention --- it would not, we believe, simply make the problem go away. In fact, we might well see buggy code such as Figure 4, Figure 7 from a professional programmer.

3. Methods for Specifying the Intention of a Program

In the above section, the basis for describing bugs was the difference between a program and the programming plans that specified a correct program. There are other methods of specifying the intention of a program:

- I/O Behavior

- Programming Plans
- Corrected Version of the Buggy Program
- Program Description Language (PDL)

In what follows we will examine each of these in turn, and explore their good points and the bad points with respect to using a method as a basis for developing bug difference descriptions.

I/O BEHAVIOR

An I/O specification for the problem in Figure 1 would be quite close to the problem statement itself. The obvious problem with this method is its vagueness with respect to the code: many different code fragments can misbehave in the same manner (e.g., there are many, many ways to generating an infinite loop --- but the I/O result is the same in all cases). One needs to be able to make finer-grain distinctions than are facilitated by a comparison of the code to simply I/O specifications.

PROGRAMMING PLANS

The major problem with this method is the need to guess what plan the programmer intended to implement. However, once the decision is made, then describing the bug as a difference between the plan and the code is relatively easy. One method of coping with the plan decision problem is interviews with the original programmers; this technique has been used to "validate" change report data in several software monitoring projects (e.g., [12]). Unfortunately, in a class of 200 students writing code at different terminals, interviews with subjects is a bit more difficult.

The major benefit derived from building a bug description using this method is an accurate reporting of the *cause* of the bug. That is, clearly the goal of a bug taxonomy in which one captures bug type and bug frequency is the ability to pinpoint the sources of the bugs: one would like to know which bugs came from misunderstandings of the specifications document and which bugs arose from coding errors, etc. For example, in the previous section if we assumed that the programmer intended to implement a ***skip-guard plan*** then we would say that there were a number of coding level bugs (e.g., WHILE instead of IF, missing ELSE, spurious READ). However, if we assume that the programmer intended to implement an ***embedded filter loop plan***, then the source of the bug may be a problem of specification interpretation: the programmer may not have thought that someone would ever input the sentinel value after inputing an illegal (negative) value. Thus he felt no need to guard subsequent computation. (An interview with the programmer would be particularly useful in this specific case.) Thus, bug categorization *and bug origin* is directly influenced by the choice of underlying plan structure in the buggy program.

CORRECTED VERSION OF THE BUGGY PROGRAM

The typical method of describing a bug is to compare the original buggy program with the corrected version of that program (e.g., [12, 7, 1]). While there is no guessing as to the intention of the original programmer, we see 2 basic problems with this approach:

- *The choice of the particular corrected program used as the measure is relatively arbitrary.* That is, there are few hard guidelines for making changes to code. Thus, different programmers could well take the same buggy program and correct it in different ways. This would result in two different bug descriptions --- an intuitively unsatisfactory situation. Moreover, different bug descriptions could lead to different conclusions as to the origins of the bugs, which, afterall, is the the point of doing the bug categorization in the first place. For example, if the buggy program in Figure 4 were corrected by implementing a ***skip-guard plan***, then the difference between the buggy program and the corrected program would result in a bug description containing 3 coding level bugs. On the other hand, if the program is corrected by putting in a guard around the subsequent computation to protect against a sentinel value, then the bug description would only contain 1 bug, a missing conditional (guard plan) --- which may or may not be a coding level bug (as discussed above). While we might prefer the programmer to make the latter change, there is no way to guarantee this situation.

  Interviewing the *original programmer* might shed some light on his intentions --- and guide the subsequent bug analysis or even bug correction. However, this additional, programmer-supplied, information goes beyond the corrected program --- and approaches a bug description based on *the programmers original plan*.While we have some methodological reservations about using interviews collected after the fact,[6] the main issue is that information gotten from the interview is of a different sort than the information gotten from the corrected program --- where the former information is much more akin to the programming plans described above.

- *What is actually counted can be quite problematic.* For example, if we correct the buggy program in Figure 7c by adding the missing ELSE, we also need to add a BEGIN-END block around the running total update and the counter update. Should we count this as 1 bug or 2 bugs? It seems unfair to count the BEGIN-END block against the programmer, since this change is required by the "real" change. On the other hand, however, in the next section we will show programs in which the "real" bug *is* a missing BEGIN-END block. Thus, it is not inconceivable that a programmer could add the ELSE in Figure 7c, but forget to put in the now necessary BEGIN-END block. What one counts is a tricky issue.

The upshot of these two problems with categorizing and counting bugs based on a corrected version of the program was suggested above: one is less confident of the origins of the bugs, and thus is less confident about percentages of bugs with those origins. Depending on the particular corrected solution and the particular choice of counting scheme, one could paint a picture of a

---

[6]The problems with using interview data has received significant attention in psychology. For example, Ericsson and Simon [4] have argued that one can reliably only use verbal information given by the subject *as the subject is doing the task*. They argue that such a concurrent verbal report is effectively an on-line dump from short-term memory. In contrast, a report after the fact could be a story about what the subject thought he was thinking, and that significant distortions can occur in this type of situation. While one might arguably feel that the Ericsson and Simon position is a bit extreme, nonetheless, it seems only prudent to exercise care in interpreting interview data.

program that contained many more coding level errors, say, than specification-based errors. The worst part of this situation is that we would not have a good way of knowing how right or wrong this analysis was --- since we don't know how the bug categories and counts would have turned out if a different corrected version were used as the basis for difference descriptions.

PROGRAM DESCRIPTION LANGUAGE (PDL)

PDL's come in all flavors; some are very close to the code, while others are more high level, and closer to the plan level description. The former PDL would suffer from the same problems as using a corrected version as the standard. The latter type of PDL would suffer from the problems associated with using the programming plans as the standard.

4. An Extended Example

Let us now consider an actual example from the on-line protocol data. In Figure 8 we depict the problem the students were trying to solve; in Figure 9 the program on the left is a buggy program generated by a student in our study. If we take a "local view" of the bugs in this program, we can generate a corrected version as shown in Figure 9 (right hand side). Notice that if we do a difference description between the corrected and the buggy versions we can come up with 8 changes:

- The rainyday counter, COUNT1, will be always be updated; in order to correct for the times when a negative rainfall is input, we need to decrement COUNT1. Thus, [1] *added a begin-end block after (NUM < 0) test, and* [2] *added a decrement of the rainyday counter.*
- COUNT2 must be made to contain the number of rainy (not just valid) days. COUNT2 keeps track of the non-rainy valid days in the loop. Thus, we need to subtract the non-rainy days (COUNT2) from the total valid days (COUNT1) in order to get the number of rainy days: [3] *changed addition of COUNT1 and COUNT2 to subtraction of COUNT2 from COUNT1.*
- The guard on the average calculation is incorrect. Thus, [4] *changed guard on average calculation to COUNT1.*
- The divisor in the average calculation should be the valid day counter, COUNT1, not the valid, but non-rainy day counter, COUNT2. Thus, [5] *changed COUNT2 to COUNT1 in the divisor of the average calculation.*
- If there is no valid input the program should neither calculate the average, nor should the program print it out --- as well as not printing out the maximum. Thus, [6] *added a begin-end block after division guard around average calculation and output statements.*
- The WRITELNs give a message about what should be output; in order to make the message agree with the actual output, the variables need to be changed: [7] *the valid day counter needs to be COUNT1, while the* [8] *rainy day counter needs to COUNT2.*

Given the number of changes that need to be made to the counters (COUNT1 and COUNT2), it would appear that the student has some confusion over the roles of the two counters.

The Noah Problem: Noah needs to keep track of the rainfall in the New Haven area to determine when to launch his ark. Write a program which he can use to do this. Your program should read the rainfall for each day, stopping when Noah types "99999", which is not a data value, but a sentinel indicating the end of input. If the user types in a negative value the program should reject it, since negative rainfall is not possible. Your program should print out the number of valid days typed in, the number of rainy days, the average rainfall per day over the period, and the maximum amount of rainfall that fell on any one day.

Figure 8: The Noah Problem: A First Looping Problem

BUGGY EXAMPLE

```
BEGIN
WRITELN ('PLEASE! INPUT AMOUNT OF RAINFALL');
READLN;
READ(NUM);
COUNT1  = 0;
COUNT2  = 0;
SUM  = 0;
HIGHNUM  = 0;
WHILE (NUM <> SENTINAL) DO
  BEGIN
  IF (NUM > 0)
    THEN
    SUM  = SUM + NUM;
    COUNT1  = COUNT1 + 1
    IF (NUM > HIGHNUM)
      THEN
      HIGHNUM  = NUM;
  IF (NUM = 0)
    THEN
    COUNT2  = COUNT2 + 1;
  IF (NUM < 0)
    THEN
    WRITELN ('ILLEGAL INPUT, INPUT NEW VALUE');
  READLN;
  READ(NUM);
  END;
COUNT2  = COUNT2 + COUNT1;
IF (NUM > 0)
  THEN
  TOTAL  = SUM/COUNT2;
  WRITELN ('AVERAGE RAINFALL WAS ',TOTAL,' INCHES PER DAY');
  WRITELN ('HIGHEST RAINFALL WAS ',HIGHNUM,' INCHES');
WRITELN (COUNT2,' VALID DAYS WERE ENTERED');
WRITELN ( COUNT1,' RAINY DAYS IN THIS PERIOD ');
END
```

CORRECTED VERSION

```
BEGIN
WRITELN ('PLEASE! INPUT AMOUNT OF RAINFALL');
READLN;
READ(NUM);
COUNT1  := 0;
COUNT2  := 0;
SUM  := 0;
HIGHNUM  := 0;
WHILE (NUM <> SENTINAL) DO
  BEGIN
  IF (NUM > 0)
    THEN
    SUM  := SUM + NUM;
    COUNT1  := COUNT1 + 1;
    IF (NUM > HIGHNUM)
      THEN
      HIGHNUM  := NUM;
  IF (NUM = 0)
    THEN
    COUNT2  := COUNT2 + 1;
  IF (NUM < 0)
    THEN
      begin                          (* add this line *)
        count1 := count1 - 1;        (* add this line *)
        WRITELN ('ILLEGAL INPUT, INPUT NEW VALUE');
      end;                           (* add this line *)
  READLN;
  READ(NUM);
  END;
count2 := count1 - count2;           (* changed this line *)
IF (count1 > 0 )                     (* changed this line *)
  THEN
    begin                            (* add this line *)
        TOTAL  := SUM/count1;        (* changed this line *)
        WRITELN ('AVERAGE RAINFALL WAS ',TOTAL,' INCHES PER DAY');
        WRITELN ('HIGHEST RAINFALL WAS ',HIGHNUM,' INCHES');
    end;                             (* add this line *)
WRITELN(count1,' VALID DAYS WERE ENTERED');   (* changed this line *)
WRITELN(count2,' RAINY DAYS IN THIS PERIOD '); (* changed this line *)
END.
```

- [1] added a begin-end block after (NUM < 0) test, and [3] added a decrement of the rainyday counter
- [2] changed addition of COUNT1 and COUNT2 to subtraction of COUNT2 from COUNT1.
- [4] changed guard on average calculation to COUNT1.
- [5] changed COUNT2 to COUNT1 in the divisor of the average calculation.
- [6] added a begin-end block after division guard around average calculation and output statements.
- [7] the valid day counter needs to be COUNT1, while the [8] rainy day counter needs to COUNT2.

Figure 9: A Buggy Program and a Corrected Version

However, consider now a different corrected version of this buggy program as depicted in Figure 10. A difference description between the buggy version and the corrected version yields the following set of bugs:

- We can make COUNT1 only keep track of the rainy days; this is consistent with code already in the program: the line that adds COUNT2 and COUNT1 now makes sense --- COUNT2 now keeps track of the valid days, and the divisor in the average calculation suggests that COUNT2 should be the valid day counter. In order to make COUNT1 perform in this manner, we need to [1] *add a begin-end pair around all computation after NUM > 0 test, up to the NUM = 0 test.*
- If there is no valid input the program should neither calculate the average, nor should the program print it out --- as well as not printing out the maximum. Thus, we need to [2] *add a begin-end block after division guard around average calculation and output statements.*
- The guard on the average calculation is incorrect. Thus, [3] *changed guard on average calculation to COUNT1.*

Which description should we choose? And why? Notice that neither of the corrected versions were that unreasonable. However, it would seem to us that one should choose the second bug description over the first. The basis for that decision is the hypothesized plan structure underlying the buggy version: it appears to us that the student was trying to structure the actions in the main loop in terms of cases. For example, the program explicitly tested for NUM > 0, NUM = 0, and NUM < 0 and took the appropriate actions --- almost. In order to make the case structure work, the code following the NUM > 0 up to the NUM = 0 test should be grouped together. While one cannot put too much faith in the indentation of a novice's program,[7] it appears that the indentation supports this analysis. Thus, what is missing from the main loop is a *begin-end* pair surrounding the code between the NUM > 0 test and the NUM = 0 test. On this analysis, the student does not have a misunderstanding surrounding the two counters, but rather has a coding level misunderstanding about how to block code together. Moreover, this same misunderstanding can explain the lack of a *begin-end* pair surrounding the average calculation in the next two write statements. The reduced bug count in the second description follows directly from this analysis: in effect there are only 3 bugs in this program, 2 of which have the same underlying origin.

This example illustrates a point made earlier: *the bug categorization and bug count follow from an understanding of the program that is provided by the hypothesized plan structure of the program.* That is, to understand a buggy program, one must make inferences about what plan structure the programmer intended to implement; the program only "makes sense" in terms of these plan descriptions.

---

[7]We have observed in the on-line protocols that the physical layout of a student's program suffers as the student makes changes to his program in the process of debugging it.

BUGGY EXAMPLE

```
BEGIN
WRITELN ('PLEASE! INPUT AMOUNT OF RAINFALL'),
READLN,
READ(NUM),
COUNT1  = 0,
COUNT2  = 0,
SUM  = 0,
HIGHNUM  = 0,
WHILE (NUM <> SENTINAL) DO
  BEGIN
  IF (NUM > 0)
    THEN
    SUM  = SUM + NUM,
    COUNT1  = COUNT1 + 1,
    IF (NUM > HIGHNUM)
      THEN
      HIGHNUM  = NUM,
  IF (NUM = 0)
    THEN
    COUNT2  = COUNT2 + 1,
  IF (NUM < 0)
    THEN
      WRITELN ('ILLEGAL INPUT. INPUT NEW VALUE'),
  READLN,
  READ(NUM),
  END,
COUNT2  = COUNT2 + COUNT1,
IF (NUM > 0)
  THEN
  TOTAL  = SUM/COUNT2,
  WRITELN ('AVERAGE RAINFALL WAS ',TOTAL,' INCHES PER DAY'),
  WRITELN ('HIGHEST RAINFALL WAS ',HIGHNUM,' INCHES'),
WRITELN (COUNT2,' VALID DAYS WERE ENTERED'),
WRITELN (COUNT1,' RAINY DAYS IN THIS PERIOD '),
END
```

ANOTHER CORRECTED VERSION

```
BEGIN
WRITELN ('PLEASE! INPUT AMOUNT OF RAINFALL'),
READLN,
READ(NUM),
COUNT1  = 0,
COUNT2  = 0,
SUM  = 0,
HIGHNUM  = 0,
WHILE (NUM <> SENTINAL) DO
  BEGIN
    IF (NUM > 0)
      THEN
        begin                          (* add this line *)
          SUM  = SUM + NUM,
          COUNT1  = COUNT1 + 1,
          IF (NUM > HIGHNUM)
            THEN
            HIGHNUM  = NUM,
        end;                           (* add this line *)
  IF (NUM = 0)
    THEN
    COUNT2  = COUNT2 + 1,
  IF (NUM < 0)
    THEN
    WRITELN ('ILLEGAL INPUT. INPUT NEW VALUE'),
  READLN,
  READ(NUM),
  END,
COUNT2  = COUNT2 + COUNT1,
IF (count1 > 0)                        (* changed this line *)
THEN
 begin                                 (* add this line *)
     TOTAL  = SUM/COUNT2,
    WRITELN ('AVERAGE RAINFALL WAS ',TOTAL,' INCHES PER DAY'),
     WRITELN ('HIGHEST RAINFALL WAS ',HIGHNUM,' INCHES'),
 end;                                  (* add this line *)
WRITELN (COUNT2,' VALID DAYS WERE ENTERED'),
WRITELN (COUNT1,' RAINY DAYS IN THIS PERIOD '),
END
```

- [1] add a begin-end pair arround all computation after NUM > 0 test, up to the NUM = 0 test
- [2] add a begin-end block after division guard around average calculation and output statements
- [3] changed guard on average calculation to COUNT1

Figure 10: A Bugggy Program an an Alternative Corrected Version

5. Concluding Remarks

We have argued that a bug description is a difference description between the realization and the intention specification. We have presented a number of techniques for specifying the intention and have pointed out the problems associated with each type of specification in developing an accurate picture of bug types and bug frequency. While no technique is without its problems, we have argued that the understanding provided by a plan analysis of the buggy program stands a better chance, as compared to the other techniques, of providing a more accurate categorization and count of the bugs --- and thus a more accurate reflection of the origins of the bugs.

References

1. Basili, V., Perricone, B. Software Errors and Complexity: An Empirical Investigation. Tech. Rept. TR-1195, University of Maryland, Dept. of Computer Science, 1982.

2. Bonar, J. Understanding the Novice Programmer. Dissertation, in preparation.

3. Ehrlich, K., Soloway, E. An Empirical Investigation of the Tacit Plan Knowledge in Programming. in Human Factors in Computer Systems , J. Thomas and M.L. Schneider (Eds.), Ablex Inc., in press.

4. Ericsson, A. and Simon, H. "Verbal reports as data." *Psychological Review 87* (1980), 215-251.

5. Johnson, L., Draper, S., Soloway, E. The Nature of Bugs in Novices' Pascal Programs. in preparation

6. Miller, L. A. "Natural Language Programming: Styles, Strategies, and Contrasts." *IBM Systems Journal 20* (1981), 184-215.

7. Ostrand, T., Weyuker, E. Collecting and Categorizing Software Error Data in an Industrial Environment. Tech. Rept. 47, New York University, Dept. of Coimputer Science, 1982.

8. Rich, C. Inspection Methods in Programming. Tech. Rept. AI-TR-604, MIT AI Lab, 1981.

9. Soloway, E., Ehrlich, K., Bonar, J., Greenspan, J. What Do Novices Know About Programming? In A. Badre, B. Shneiderman, Ed., *Directions in Human-Computer Interactions*, Ablex, Inc., 1982.

10. Soloway, E., Bonar, J., Ehrlich, K. . Cognitive Strategies and Looping Constructs: An Empirical Study. Communications of the ACM, in press.

11. Soloway, E., Rubin, E., Woolf, B., Bonar, J., Johnson, L. MENO-II: An Intelligent Programming Tutor. Journal of Computer-Based Instruction, to appear.

12. Weiss, D. Evaluating Software Development By Analysis of Change Data. Tech. Rept. TR-1120, University of Maryland, Dept. of Computer Science, 1981.

THE VIEWGRAPH MATERIALS

for the

W. JOHNSON/S. DRAPER/E. SOLOWAY PRESENTATION

WERE INCORPORATED IN THE PAPER

D10

N83 32366

ERROR TAXONOMY

WHAT CAN BE GAINED?

by

D. E. Buckland

Reifer Consultants, Inc.

December 1, 1982

System development has been and continues to be an evolutionary process. Technology is rapidly catching up with the science fiction writers of yesterday. We see some form of computer in just about all of our equipment, including cars, watches, cameras, home appliances, weapons systems, communications devices, space ships, etc. In the good old days, hardware did it all. Today more and more capabilities are being fashioned by some form of software, and computers are becoming smaller, more powerful and far more complex. We've recognized that, no matter what our task is, experience is our best teacher. In the field of system development we'd like to profit not only from our own experiences, but also from the experiences of our fellow computer scientists.

In order to do this, we need a history of what we've done. We can accomplish this by implementing some of the formal procedures and documentation requirements from the older, hardware side of the house. In order to profit from our mistakes, we need to keep track of what went wrong, and what was done to correct each situation. One technique used to accomplish this is to implement an error taxonomy.

Exactly what is an error taxonomy? Simply stated, it is the classification and quantification of errors. Numerous studies have been conducted in an attempt to provide quantitative data on errors that occurred in relatively large systems. The study of errors is important for the following reasons:

- A major item impacting costs, risks and uncertainty in system development is the lack of knowledge of what causes errors, why they occur and how they can be reduced (or at least located more quickly). The development of error data bases for systems is a step towards the statistical quantification of error occurrence. Once error occcurrences can be quantified, steps can be taken to reduce them.

- Identification of relationships between error occurrences, causes, criticality and time of error occurrence can lead to improved methods of detecting errors before they become difficult and costly to correct.

- Reliable error data can be used to measure the impact (both positive and negative) of modern software development and validation methodologies and tools on quality and productivity.

- The formal error documentation process forced by error data collection itself can provide better error control and help assure appropriate corrective actions are taken.

Errors can be categorized in a number of ways. The key is to define categories that are useful and applicable to the application. The more common categories are:

- o Time of occurrence
- o Level of criticality
- o Error type
- o Time of introduction

The main reason for reporting problems is so that each problem can be resolved in a timely fashion. During system development and subsequent use, problems are found and reported regularly. If a formal reporting process is not used, even in a one man job, some problems fall by the wayside, and linger to make themselves known at some inconvenient time in the future. Programmer X discovers a problem in programmer Y's code, and with full intentions of telling him (or her) about it as soon as his test time is finished, becomes involved in another problem, or runs off to a meeting, and forgets. Or how many times have we heard "Such and such doesn't work correctly" with no indication of what was being done or what was expected? Much time and effort must then be expended to investigation prior to resolution.

In order to identify and solve problems in a timely fashion, a clean, simple problem reporting mechanism is required. Using such a mechanism, problem status reports can be produced that enable management and staff alike to evaluate what is left to be done, assign priorities so that the more painful items are taken care of first and group similar problems together for expeditious handling. When problem reports are up to date, test coverage can be maximized by staying clear of known problem areas, concentrating on new territory and reducing duplication. When thorough problem reports are required, test objectivity increases because test conditions must be substantiated. The problem report itself serves as a form of communication between reporter and resolver, and problem turnaround increases. A careful analysis of problem status reports can identify weak areas, spot trends and enable the application of past experiences in the future.

The reporting mechanism must include the filling out and gathering of problem reports, enable expedient investigation, archive the resolution and enable problem evaluation. All of this should be accomplished with a minimum of clerical time. A key point to remember is to gather enough data at the time so that information you may need in the future is readily available.

When implementing a problem reporting system, several factors need to be considered beforehand. The first is to define a common set of terms so that all involved with the system are speaking the same language. Establish and publish a list of keywords, acronyms and abbreviations. Next one should design a problem reporting form. This should be kept to one page and should make use of checkboxes where practical. Plenty of space should be provided for both the problem symptom and the resolution. Allow for problems to be reported against a baseline, with all deviations from the baseline noted (patches, etc.). One central point of control should be maintained, where new problems can be logged open, and resolved problems closed.

This may be as simple as a notebook or as complex as an automated system. Of prime importance is to assure that the system is flexible and growth oriented. It is much easier to gather data in real time than to acquire it from the memories of those involved when the project is completed.

Information on problems is usually collected in serial fashion. When a problem is discovered, the following is needed:

- Who found the problem? Should a question arise as to the nature of the bug, facts not included in the report itself, interpretation of the test, recreation of the problem, etc., it will be necessary to speak with the reporter.

- When was this problem found? Recording this date enables the analyst to arrive at such facts as what phase of the life cycle this occurred in, how long the problem has been open and how long it took to resolve, and also to track how many problems were opened during given phases.

- What happened? The reporter should detail the exact symptoms whenever possible. This includes, but is not limited to, the system identification, hardware and software configurations, test case, inputs, test programs, expected outputs or reactions, etc. There should be enough detail to enable the programmer to recreate or pinpoint the problem. Remember, it is entirely possible that one problem can have several symptoms.

- What was being used? The system the problem occurred on, along with any test equipment should be identified. This will enable the programmer to determine whether the problem is configuration dependent, or possibly caused by a hardware failure.

- Is this a reoccurrence of a previously closed problem? This would indicate that a problem may have occurred in configuration management, or all of the causes had not yet been discovered.

- What is the level of criticality? The category must take into consideration whether or not the problem itself is mission critical, prevents further checkout of mission critical areas of the system, will involve a lot of rework and impact schedule, is cosmetic in nature, etc. The level of criticality is not always evident when the problem is originally reported, but may change as investigation reveals the mitigating conditions.

When a problem is resolved, the appropriate historical information should be recorded. Analysts will need to know:

- Why did it fail? The clinical reasons for the failure must be recorded. The modules and interfaces involved should be noted. The exact cause should be given, whether it was an

error or oversight in the requirements, a design failure, coding error, test error, human operation fault, etc. This information will allow the analyst to identify error trends and weak areas, and suggest recovery actions.

- What was the solution? Exactly what was done to resolve the problem? This might be to correct a piece of documentation, revise the code, or even do nothing at all. Depending on when a problem is found, it is sometimes more costly and more risky to fix it than to work around it.

- Who supplied the resolution? Should questions arise in the future, this is the person to whom they will be directed.

- When was it closed? The presence of this date indicates that the problem is not active, and will not be included in the "current open" count. It also enables time information to be extracted.

During the time that a problem is open, it may prove helpful to give it a status, such as new, patched, reported fixed on a certain baseline, retry, recreate, revised, etc. These can indicate to those using the report actions that need to be taken to close the problem. For instance, a problem that is categorized as critical, but has not been reproducible, would carry a recreate status to indicate that the programmer wishes to be informed immediately when the problem re-occurs. Or a problem reported as fixed on a given baseline should be validated prior to its being officially closed.

Now that we have the ability to collect all of this fine data, what can it tell us? By way of example, let me share with you a study that was conducted by Reifer Consultants, Inc. of errors reported during the development and use of the Deep Space Network/3 in preparation for the development of the Deep Space Network/4.

The problem reports for this program were initially meant to indicate to the programmers that a problem existed, and not much more. In preparation for this study, a team of analysts evaluated existing taxonomies, and with a little embellishment, developed a taxonomy applicable to this JPL project. A three dimensional classification scheme was devised to capture meaningful error data in a manner suitable for additional statistical and trend analysis. Each of the dimensions is summarized below:

- Time of Occurrence - Defines in which of the four DSN phases of the software life cycle the error occurred. The four times were: Development, Verification, Acceptance or Transfer.

- Criticality - Defined in which level of severity the error could be categorized. The three levels of severity were: Critical, Dangerous and Minor.

- o Category - Categorized the cause of the error. The ten error types were: Computation, Logic, Data handling, Interface, Data base, Operation, Requirements incorrect, Design, Clerical and other.

(Because it is important to precisely define terminology, I have enclosed a detailed description of the taxonomy as an appendix to this paper.)

The same team of analysts then analyzed approximately 1000 problem reports, and interviewed people involved with the projects in an attempt to fill in the blanks. Using the DSN/RCI software error taxonomy, each problem report was categorized in terms of its category, criticality and time of occurrence.

A preliminary analysis of the resulting data base was performed. Summaries of the data were compiled and evaluated so that recommendations for improvement could be formulated. Histograms were used to identify apparent trends and conclusions without resorting to a detailed statistical analysis. The histograms combine error data within accuracy range of plus or minus 1%. Three histograms follow along with a discussion of the observations. To simplify the graphs, the common abbreviations listed in Table 1 were used.

Table 1
ABBREVIATIONS/ACRONYMS

o Time of Occurrence

D - Development - design, coding and unit test
V - Verification - integration and testing of subsystem
A - Acceptance - Formal testing and acceptance of subsystem
T - Transfered - software subsystem operational
U - Unknown

o Criticality Levels

A - Critical
B - Dangerous
C - Minor
U - Unknown

o Error Category

CO - Computational Error
LO - Logic Error
DH - Data Handling Error
IN - Interface Error
DB - Data Base Error
OP - Operation Error
RI - Requirements Incorrect
DE - Design Error
CL - Clerical Error
OT - Other

A histogram illustrating errors by time of occurrence (Figure 1) was produced. The undefined time occurrences resulted from problem reports which had no time of occurrence and for which no time of occurrence could be ascertained. The observations we can make based on this histogram are as follows:

- o The data seems to indicate that formal problem reporting procedures were not strictly enforced during the development of most of the subsystems investigated by this study.

- o The software verification and acceptance testing processes uncovered a large number of errors. Unfortunately, there were still many more errors not discovered until the subsystem was placed in operation.

The next histogram (Figure 2) illustrates errors by criticality level for each of the three criticality indices. An additional U classification was included to identify anomalies for which no criticality level could be ascertained. The observations we can make based upon this histogram are as follows:

- o Level B errors were in the majority. Although work arounds could be devised, such a large number of errors makes existing quality assurance practices suspect.

- o A large number of level A errors were identified. Critical errors of such a large proportion immediately call attention to review procedures and testing approaches used during development.

The next histogram (Figure 3) illustrates criticality level by error category. An additional classification, "questionable", consists of "other" problem reports for which no change was generated. These "questionable" errors were the subset of "other" errors which resulted from documentation requests, gripes, misunderstandings, politics and potential hardware failures. The observations we can make based on this histogram are as follows:

- o Design errors seemed to cause a large number of critical errors. This provided us with further evidence of the need to investigate earlier detection of design errors.

- o Data handling errors were also a cause of a large number of critical errors.

- o Surprisingly, "other" errors contributed a large number of critical errors. This could be attributed to the user who could not operate or understand operational anomalies and categorized them as critical to get immediate attention. This data emphasized the need to revamp the existing problem reporting procedure and to investigate ways of improving the man/machine interface.

FIGURE 1

ERRORS BY TIME OF OCCURRENCE

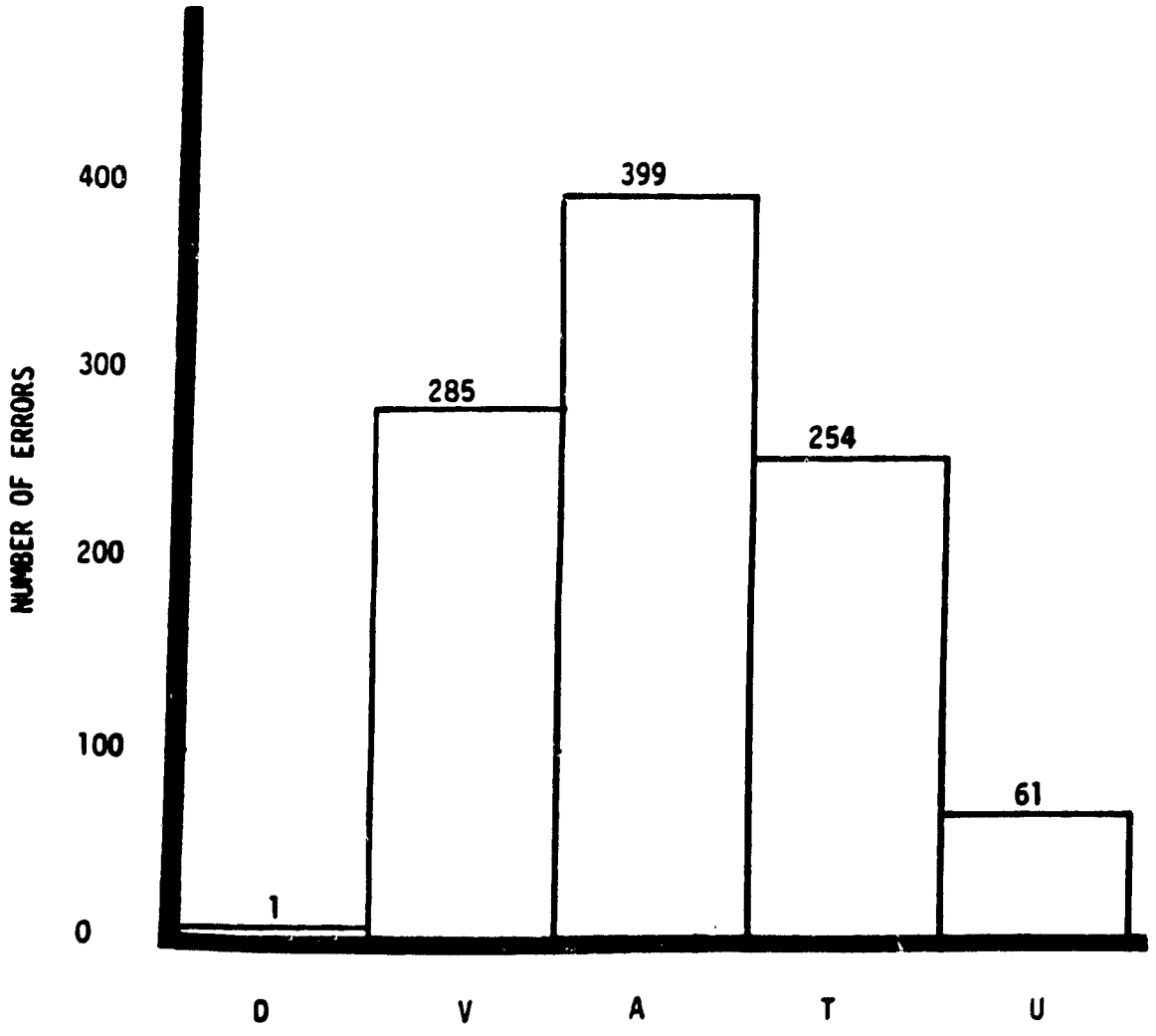
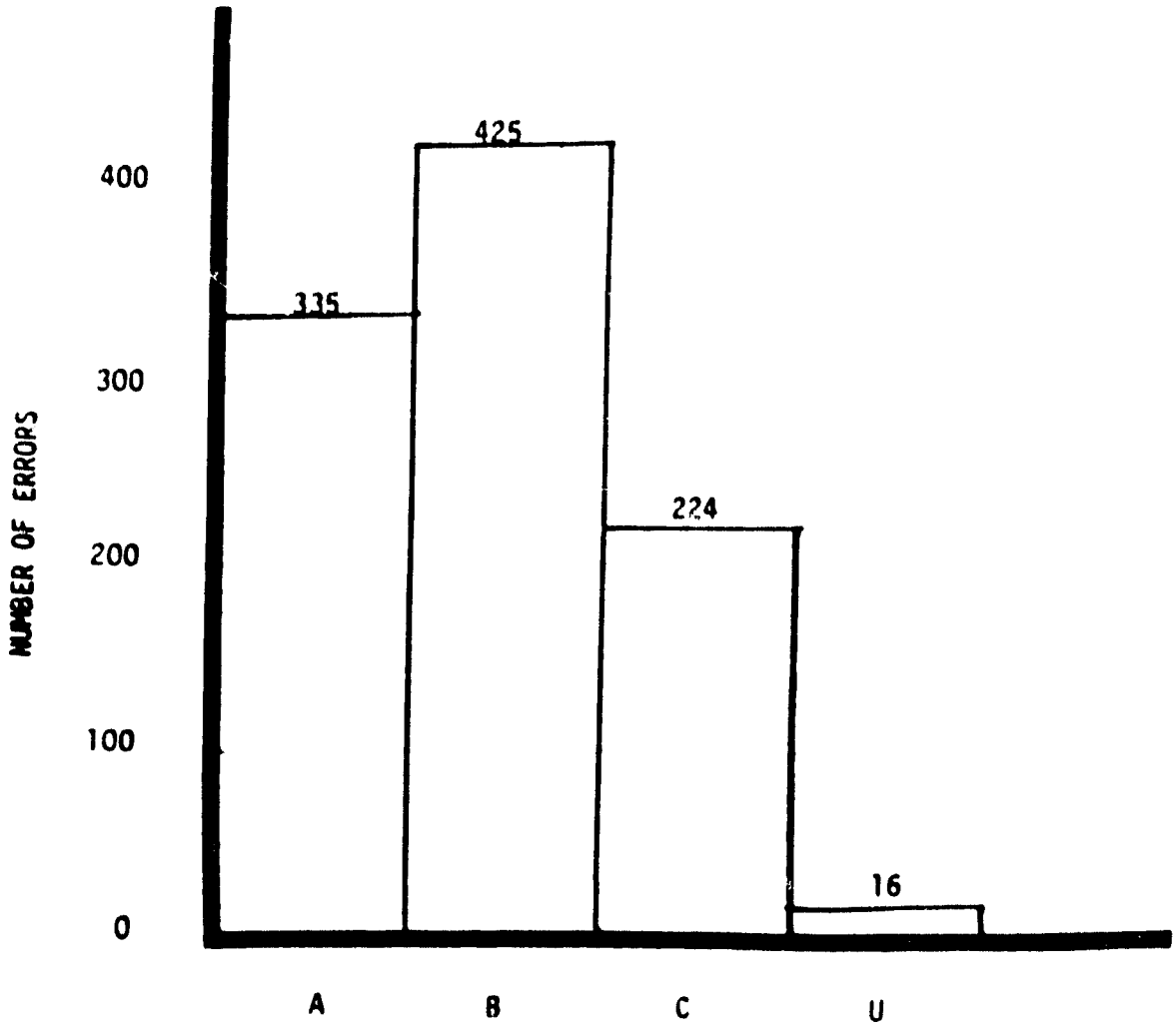


FIGURE 2

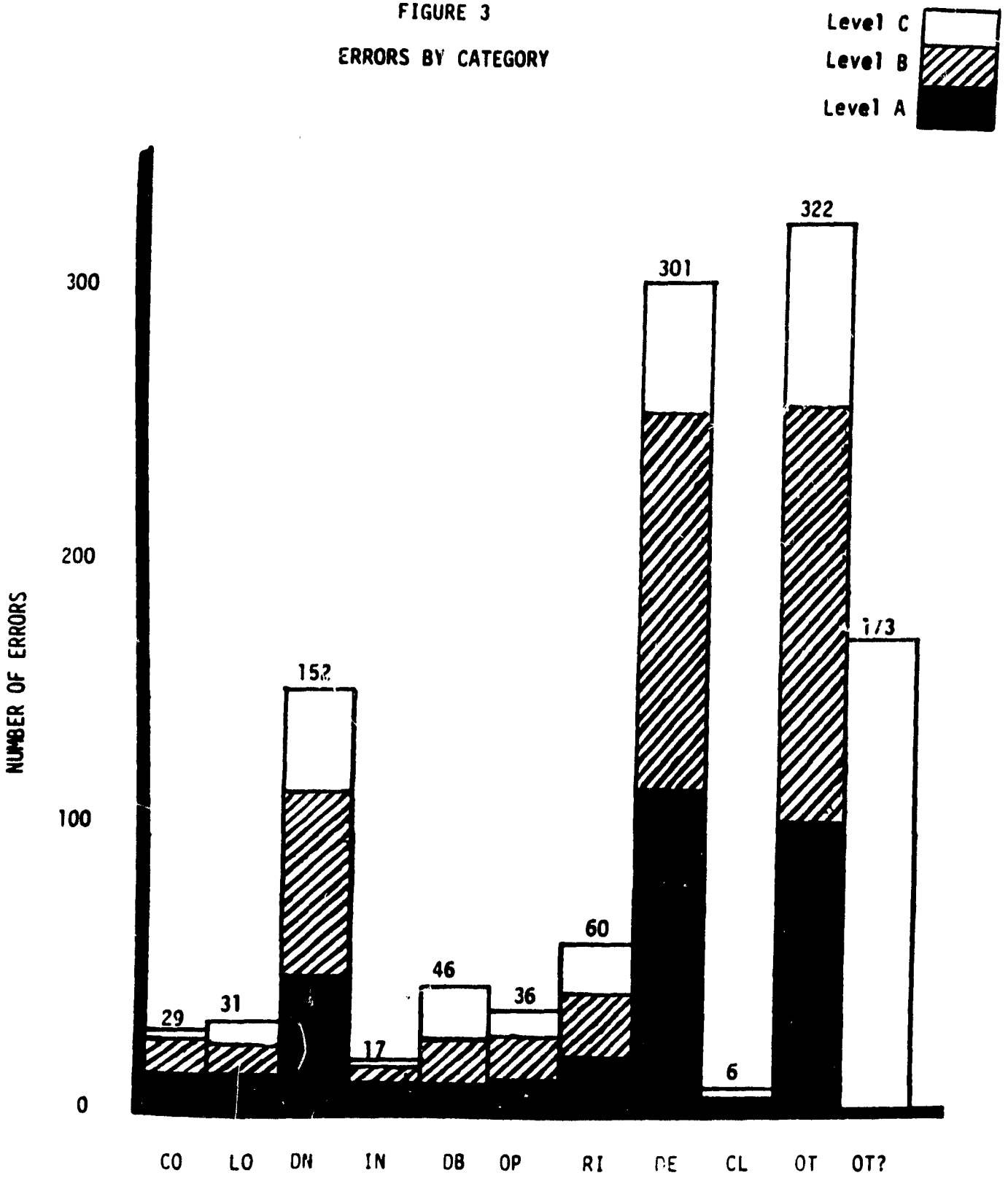
ERRORS BY CRITICALITY



ORIGINAL PAGE IS
OF POOR QUALITY

FIGURE 3

ERRORS BY CATEGORY



- o Design and requirements errors were the largest single source of problems.

- o Some errors of the "questionable" subcategory of "other" were not errors but really requests for changes or documentation. This seemed to indicate the need to improve existing problem reporting procedures and the mechanisms used for quality control.

The major findings of this study can be summarized as follows:

- o Software error data is an important management tool because it indicates where problems exist and where management attention should be placed. For future projects, the classification of error data should be performed as anomalies are reported. This would help assure that the error was more fully understood as it was reported. It could also be used to identify error-prone modules and provide information upon which repair or replace decisions could be based.

- o Analysis of the DSN software error data base indicated that many of the critical errors occurred during the requirements definition and design phases. These errors are the most costly to correct, especially if they are not caught early in the development cycle.

- o Many of the "other" error types could be attributed to poorly defined man/machine interfaces (e.g., commands that are difficult to use or whose incorrect usage causes the system to halt), improper and imprecise procedures for handling exceptions, inadequate documentation and/or user misconceptions (requestsd for enhancements/modifications that were not really problems at all).

ACKNOWLEDGEMENT

Portions of this paper are based upon work performed by Reifer Consultants, Inc. under Contract L0-726925 to the Jet Propulsion Laboratory, California Institute of Technology. It utilizes Deep Space Network anomaly data compiled by Ms. Connie Johnsen and analyzed by SoHaR, Inc. under subcontract to RCI. Many people supported our efforts and all of their contributions are acknowledged. Special thanks are extended to the DACS at Rome Air Development Center who has agreed to distribute the error data base free to interested parties.

Appendix

Software Error Taxonomy Definitions

Time of Error Occurrence

Four time classifiers were chosen because they were compatible with the DSN anomaly report data provided as input. The classifiers are as follows:

(D) Development - Anomalies in this category were reported during the design, coding and module unit testing activities. Most required design or programming revisions to be made. Errors in the category typically dealt with design problems between modules or with functional limitations of design. An example follows:

> "A system was required to provide human readable error messages on a log device. Unfortunately, the function was not specified in either the requirements and design specification. The error was discovered during a design review and an anomaly report was opened. Under such circumstances, we would state that the anomaly had occurred during development."

(V) Verification - Anomalies in this category were reported during integration and testing activities. Most were specification deviations that required the code to be revised. An example follows:

> "Module X expects a true or false condition as input from module Y. Unfortunately, module Y has not been specified to provide the true or false input. A test identified this problem during testing and an anomaly report was written scoping the rework. Under such circumstances we would state that the anomaly had occurred during verification."

(A) Acceptance - Anomalies in this category were reported during formal testing of the software. Errors in this category usually stem from requirements problems or improper mechanization. An example follows:

> "The system malfunctions when accepting more than six simultaneous inputs. The error was discovered during formal testing when the program was stressed and an anomaly report was written. Under such circumstances, we would state that the anomaly had occurred during acceptance."

(T) Transfer - Anomalies in this category were reported after the software package was put into operation in a live environment. These anomalies usually resulted from halts, failures or malfunctions. An example of such an anomaly follows:

> "The software halfts when a zero input value is received. This error was discovered during operation when the DSN was reducing telemetry data. Under such circumstances, we would state that the anomaly occurred during transfer."

Error Criticality

The three error criticality classifiers used are defined as follows:

- Level A - Critical error (error impacts mission performance or seriously degrades capability and no workaround exists). An example follows:

> "The system halts when the value of one of its inputs exceeds its nominal end of range. Manual intervention is required before operation can be resumed. Under such circumstances, we would state that a level A error had occurred."

- Level B - Dangerous situation (error exists that could degrade performance or capability but a workaround exists). An example follows:

> "A particular utility function causes the system to halt to await operator's action. The utility function is not required for correct system operation and can be disabled temporarily to correct the problem. Under such circumstances, we would state that a level B error had occurred."

- Level C - Minor problem (error exists that doesn't impact performance or capabilities and can be fixed at a more leisurely pace. An example follows:

> "An informational message is displayed twice (rather than once) each time it is enabled. No other negative effect happens. Under such circumstances, we would state that a level C error had occurred."

Error Category

The third dimension of the DSN/RCI error taxonomy is error category. Each of the ten error categories was defined so that insight into the error causes could be ascertained. The ten categories are defined as follows:

1. Computation - Computation anomalies are errors in or resulting from coded equations. Examples of computation errors include: (a) Incorrect operand in equation, (b) Incorrect use of parenthesis, (c) Incorrect equation, (d) Missing computations and (e) Rounding or truncation error.

2. Logic - Logic anomalies are errors in sequencing, control or loop conditions. Examples of logic errors include: (a) Logic out of sequence, (b) Wrong variable being checked, (c) Missing logic or condition tests, (d) Too many/few statements in loop and (e) Loop iterated incorrect number of times.

3. Data Handling - Data handling anomalies are errors in handling input/output. Examples of data handling errors include: (a) Data initialization incorrect, (b) Variables not set properly, (c) Variable type incorrect, (d) Data packing/unpacking incorrect and (e) Subscripting error.

4. Interface - Interface anomalies are errors in communications between a routine and other routines, the data base and/or the user. Examples of interface errors include: (a) Data incorrectly transmitted from one routine to another, (b) Data incorrectly set/used from the data base, (c) Improper input/output synchronization and (d) Data sent to wrong destination.

5. Data Base - Data base anomalies are errors in present data. Examples of data base errors include: (a) Data should have been initialized in data base but wasn't, (b) Data initialized to incorrect value and (c) Data base units are incorrect.

6. Operation - An operation anomaly is an error occurring as the software executes. Examples of operation errors include: (a) Operating systems errors, (b) Hardware errors, (c) Operator errors, (d) Compiler or support software errors and (e) Test execution errors.

7. Requirements Incorrect - Requirements errors deal with improper or ambiguous functional and software requirements specifications and not with implementation and/or operation. Software may correctly solve the wrong problem if it is specified improperly.

8. Design - Design errors deal with improper architectural and detailed design specifications which form the basis to which the program and the data base are mechanized.

9. Clerical - Clerical anomalies occur when people are involved in the translation. Examples of clerical errors include keypunch, typos and/or transliteration.

10. Other - Other is a "catch-all" for other types of errors not encompassed by the scheme. Examples of other errors include incorrectly reporting that an anomaly had occurred when in reality it was a programmer misconception.

THE VIEWGRAPH MATERIALS

for the

D. BUCKLAND PRESENTATION FOLLOW

ERROR TAXONOMY

WHAT CAN BE GAINED?

by

D. E. Buckland

Reifer Consultants, Inc.

25550 Hawthorne Boulevard, Suite 208/Torrance, California 90505

Reifer Consultants, Inc.

WHAT DOES IT DO?

AN ERROR TAXONOMY IS A TOOL
THAT ENABLES US TO BETTER
LEARN FROM OUR MISTAKES

WHY REPORT PROBLEMS?

SO THEY CAN BE RESOLVED!

PROBLEM STATUS REPORTS

- Aid In the Evaluation of What You Have Left To Do
- Group Similar Items Together For Expeditious Handling
- Assign Priorities
- Increase: Test Coverage
  Objectivity
  Communication
  Turnaround
- Reduce Time & Paperwork
- Learn From Past Experiences
- Identify Weak Spots
- Spot Trends

WHAT IS THE MECHANISM?

A CLEAN, SIMPLE PROBLEM REPORTING SYSTEM

- Problem Reports
- Problem Investigation
- Problem Resolution
- Problem Evaluation

} Individual Basis

HOW DO I DO IT?

RECORD ENOUGH DATA TO GET AT THE INFORMATION YOU MAY NEED LATER

- BUILD IN FLEXIBILITY
- REPORT BY BASELINE
- USE CHECKBOXES WHERE POSSIBLE
- PLAN AHEAD
- AUTOMATE
- ONE CENTRAL POINT OF CONTROL
- COMMON TERMINOLOGY

Reifer Consultants, Inc.

WHAT WILL I NEED TO KNOW?

AT TIME OF DISCOVERY:

- WHO FOUND IT?
- WHEN WAS IT FOUND?
- WHAT HAPPENED?
- WHAT WAS BEING USED?
- IS THIS A REOCCURRENCE OF A CLOSED PROBLEM?
- WHAT IS THE LEVEL OF CRITICALITY?

Reifer Consultants, Inc.

WHAT WILL I NEED TO KNOW? (cont)

AFTER THE PROBLEM HAS BEEN CLOSED:

- WHY DID IT FAIL?
- WHAT WAS THE SOLUTION?
- WHO ASSIGNED THE RESOLUTION?
- WHEN WAS IT CLOSED?

ORIGINAL PAGE IS
OF POOR QUALITY

WHAT CAN I DERIVE?

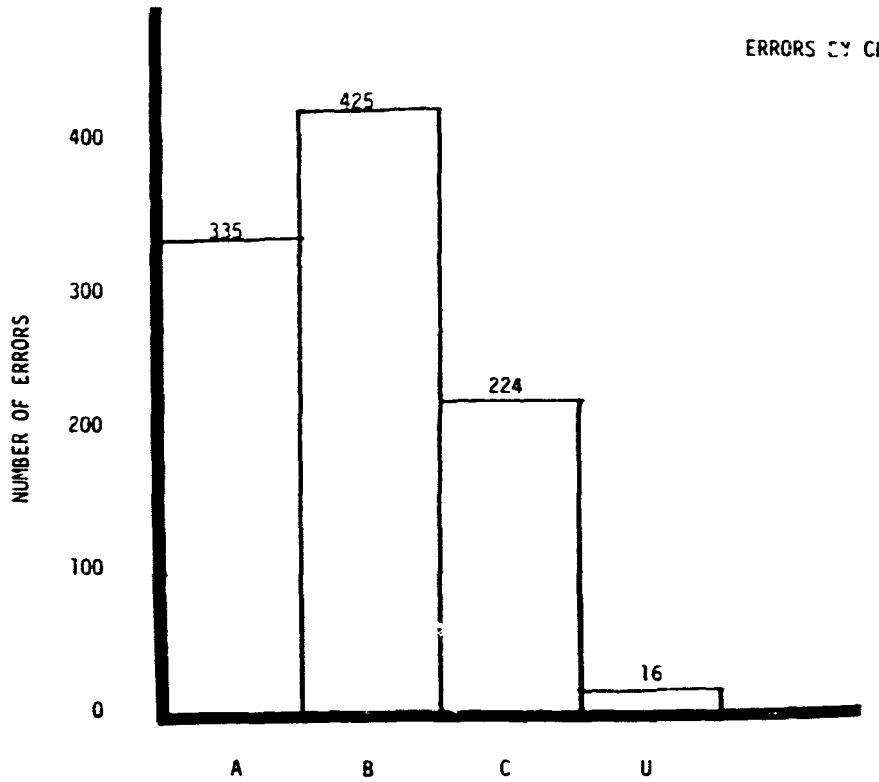
- PROBLEM OCCURRENCE RATES
- RESOLUTION RATES
- INFORMATION BY CATEGORY
  - System
  - Subsystem
  - Module
  - Criticality
  - Baseline
  - Problem Type
-
-
-

AN ANALYSIS OF ~1000 SOFTWARE ERRORS FROM DSN/3

Study Conducted by RCI for JPL in 1981

Reifer Consultants, Inc.

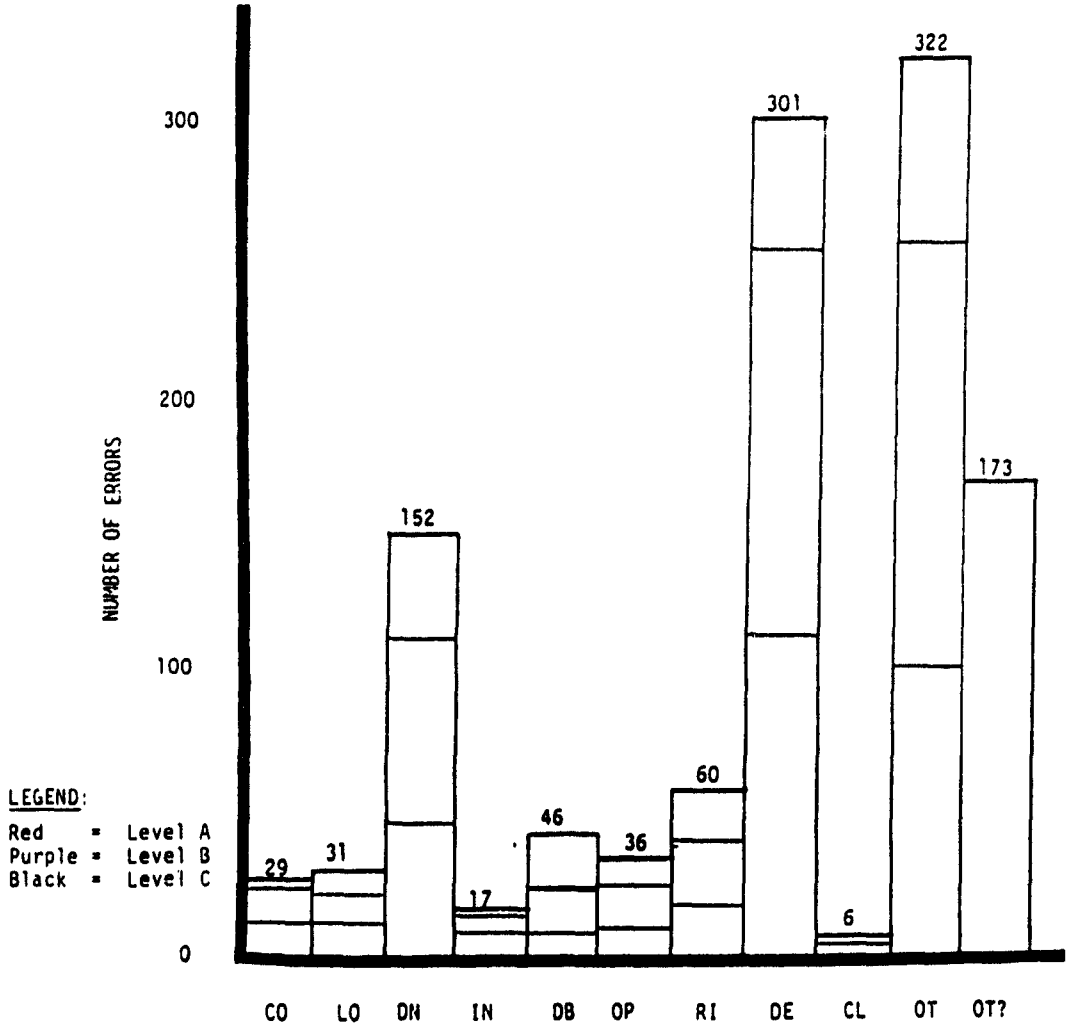
DSN-3



ORIGINAL PAGE IS
OF POOR QUALITY

Reifer Consultants, Inc.

DSN/3



Reifer Consultants, Inc

SUMMARY

THE QUANTIFICATION OF ERROR DATA IS AN IMPORTANT MANAGEMENT TOOL

- Where You Are
- What You Did Right
- What You Did Wrong

PANEL #4

COST ESTIMATION

K. Rone, IBM
R. Tausworthe, JPL
R. Britcher J. Gaffney, IBM

D11

N83 32367

MAINTENANCE ESTIMATION METHODOLOGY

BY

KYLE Y. RONE

INTERNATIONAL BUSINESS MACHINES CORPORATION

FEDERAL SYSTEMS DIVISION

HOUSTON, TEXAS

INTRODUCTION

As a project nears the end of its development phase and prepares to enter a maintenance phase, several questions arise for which there are no ready answers:

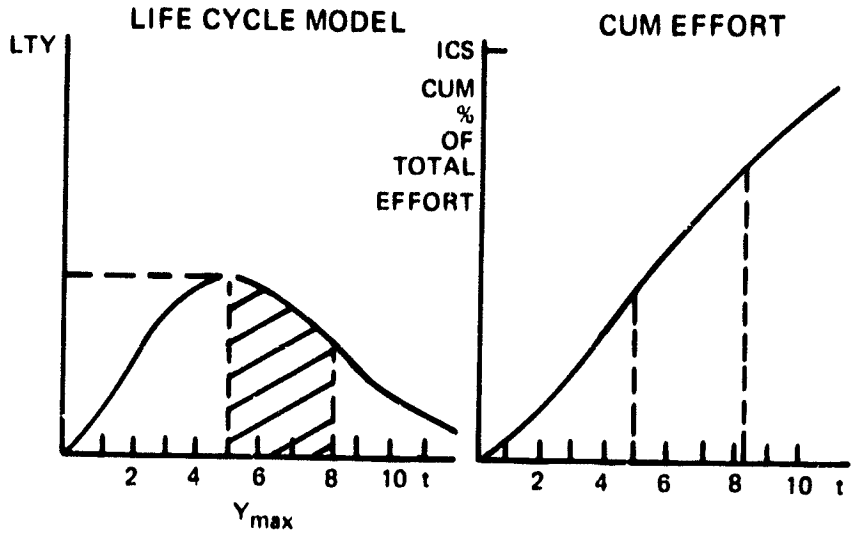
- How many people are required to maintain the system?
- What is the required critical skills level to support the project?
- What is the required staffing level to be responsive to customer needs?
- How much of the staffing level can be used to perform new development work?

The purpose of this paper is to develop a rational, systematic approach to answering these questions. The approach selected uses a Rayleigh curve method of projection combined with a modified matrix method to forecast maintenance needs and required staffing levels. The curves generated by both methods are differenced to ascertain how much new work can be performed given the staffing line. Finally, actual project data is compared to the projection to validate or modify the process.

DETERMINING MAINTENANCE NEEDS

In order to determine maintenance needs in the future, it is first necessary to examine the entire software development process. Studies by Peter Norden of IBM (Reference 1) have shown that research and development projects are composed of cycles. When these cycles are related to one another and added together, a curve results which represents the entire project. Furthermore, these curves can be approximated by the Rayleigh curve forms given in Figure 1. Since software systems follow a life cycle process similar to other research and development projects, the Rayleigh curve method is selected for use in this methodology.

To use this method, the foregone development phase is examined for actual manpower expenditures. A Rayleigh curve is then generated which approximates the curve of the expenditures during the development process. The resultant curve beyond the delivery point of the software system represents a projection of manpower needs during the maintenance process which is driven by the work expended during the development process.



$Y' = 2Kate^{at^1}$

$Y = K(1-e^{at^1})$

IMPORTANT PARAMETERS:

K = TOTAL MY FOR ENTIRE PROJECT

$K = e \cdot \frac{1}{2} \cdot Y'_{max} \cdot t_{y'max} = \sqrt{e} \; y'max \cdot t_{y'max}$

$a = \frac{1}{2} t^2_{y'max}$

DETERMINING A REASONABLE LEVEL OF SUPPORT

The Rayleigh curve method, then, projects future work based on past work. This method however is based on pure work required and does not address other project needs as critical skills and response to software system problems. Given that the development work stops at some point, then the curve will eventually go to zero. whereas, as long as software support is required, the project will continue to supply it. A method is required, then, to determine a reasonable level of software development support to be provided to the customer at some steady state period in the future.

To accomplish these goals, a study is performed across the software project to determine functional elements and drivers for each project area. These functional elements and drivers are then used to develop a matrix approach to estimating support levels for each project area. Each element is then quantified by software size, number of test cases required, or by development manpower level. These quantifiers are then transformed into maintenance levels for the element by use of the following general equation:

$$\text{Maintenance Level} = \frac{\text{ELEMENT SIZE}}{(\text{Productivity})(\text{Complexity Factor})(\text{Level Factor})}$$

Where:
- Productivity = development or test productivity factor
- Complexity Factor = varies about .5 based on the complexity of the element
- Level Factor = 12 (length of development)

The resultant maintenance levels are then tempered and modified based on judgments concerning critical skills and operations support and the totals are increased by a fixed percentage to cover management and support. An example of a matrix for a given area of software is depicted in Figure 2. All areas are summarized for the project to determine the required support level (Figure 3). This generated level can be plotted with the Rayleigh curve as shown in Figure 4. The Rayleigh curve represents current effort required based on past effort. The optimal staffing level to be reached in steady state is represented by the support line.

SM

FUNCTION	STS-1 SIZE	STS-2 SIZE	DEV. LEVEL	MAINT. LEVEL	CRITICAL SKILLS	OPN SUPPORT	TOTAL SUPPORT
SM BASIC	10875	11792	3.2	1.4			1.4
SM/SP	4657	4709	1.3	.6			.6
SM-DISP CONT.PROC.	9109	12041	3.2	1.4			1.4
DOWNLIST	10106	10106	2.7	1.2			1.2
RMS	45	7574	2.0	1.0	1.0		2.0
SM ROLL INS	8658	8658	2.3	1.1			1.1
SM,DL,ANNUN. PREPROC.	-	-	8	1.5 1.5		1.0 2.0	3.0

FIGURE 2. EXAMPLE OF AREA MATRIX

MATRIX ESTIMATE SUMMARY

AREA	SIZE	MAINT. LEVEL	OPN & SUPPORT	M&S	TOTAL
AASD	272918 FW	42.0	12.0	10.0	64.0
AASD	43318 FW	31.9	8.0	10.1	50.0
CON/QA	-	5.0		1.0	6.0
SEC. SUPP.		11.0			11.0
SDL	875K S/L	36.0	4.0	7.0	47.0
ASVO	1247 TC	82.5	5.0	15.5	103.0
SAS M&S	-			4.0	4.0
		208.4	29.0	47.6	285.0

FIGURE 3. EXAMPLE OF MATRIX ESTIMATE SUMMARY

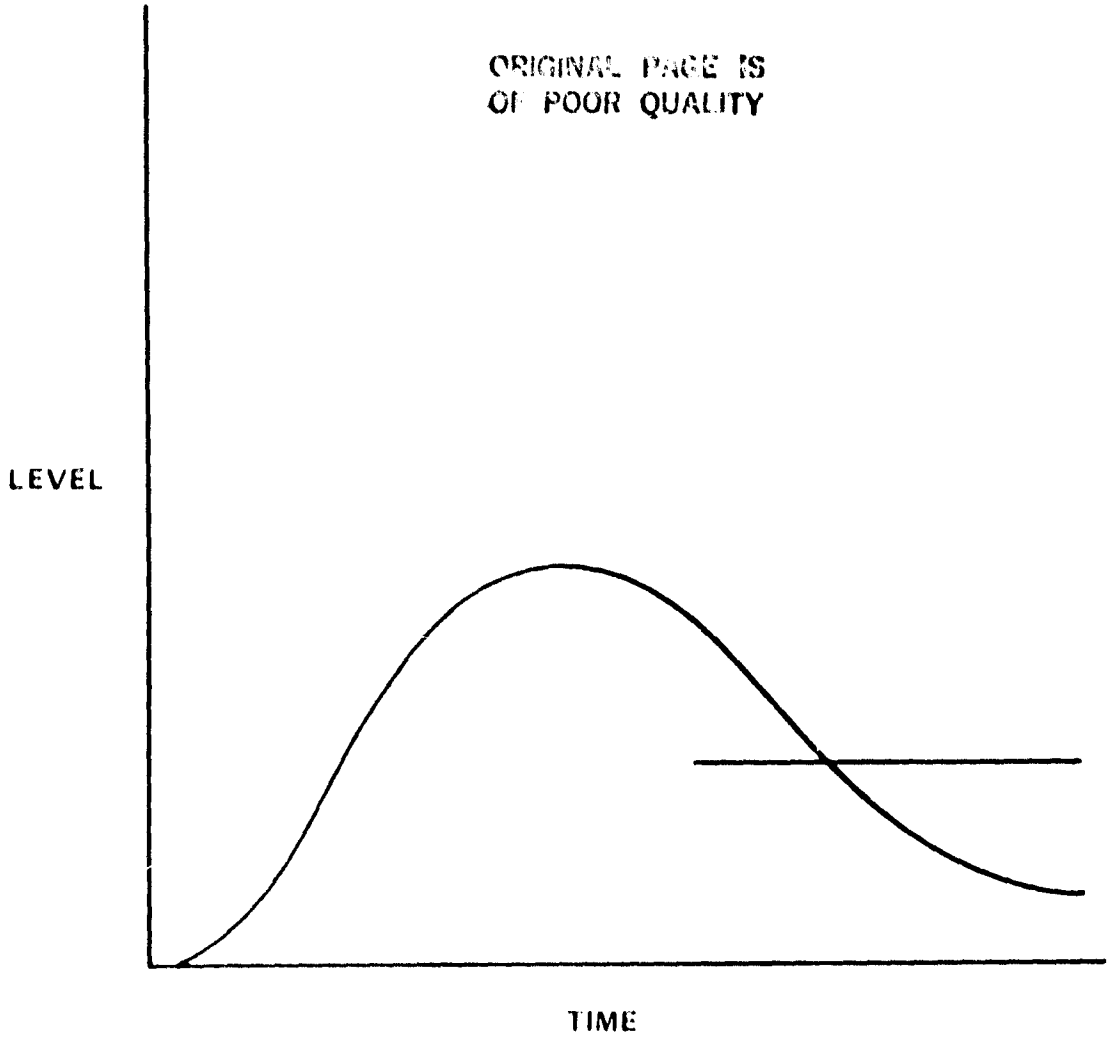


FIGURE 4. PLOT OF RALEIGH AND SUPPORT LINE

MANPOWER AVAILABLE TO PERFORM NEW WORK

The plot of the Rayleigh curve and the support line can also be represented as two equations. By integrating the difference between the two equations and evaluating over the time of interest, the area between the curves is generated. This area represents the amount of manpower supported by the staffing level which is not committed to maintenance of past work, and hence, can be applied to new tasks (Figure 5).

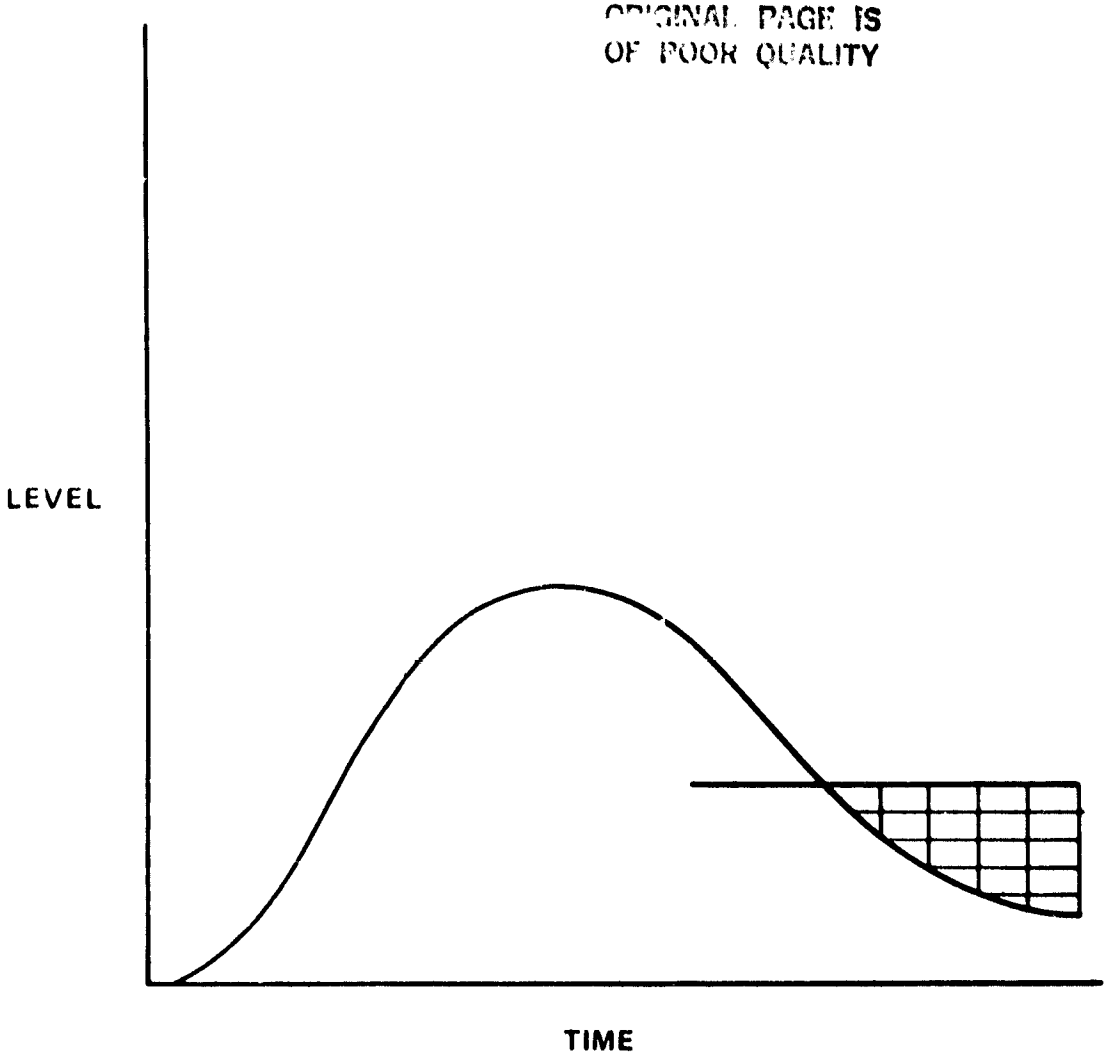


FIGURE 5. MANPOWER AVAILABLE TO PERFORM NEW WORK

CONVERTING DIRECT ESTIMATES TO TOTAL PROJECT COSTS

Using the manpower available to perform new work requires that direct work estimates be converted to project costs consistent with the project costs represented by the curves. To derive this relationship, examine the direct costs and overhead costs from actual data and calculate:

$$\text{PROJECT FACTOR} = \frac{\text{Total Project Cost}}{\text{Direct Estimate}}$$

Using this factor, an estimate for a change or group of changes can be turned into a total project cost and used to "fill up" the area between the curves (Figure 6) until the project's capacity to perform new work is exhausted.

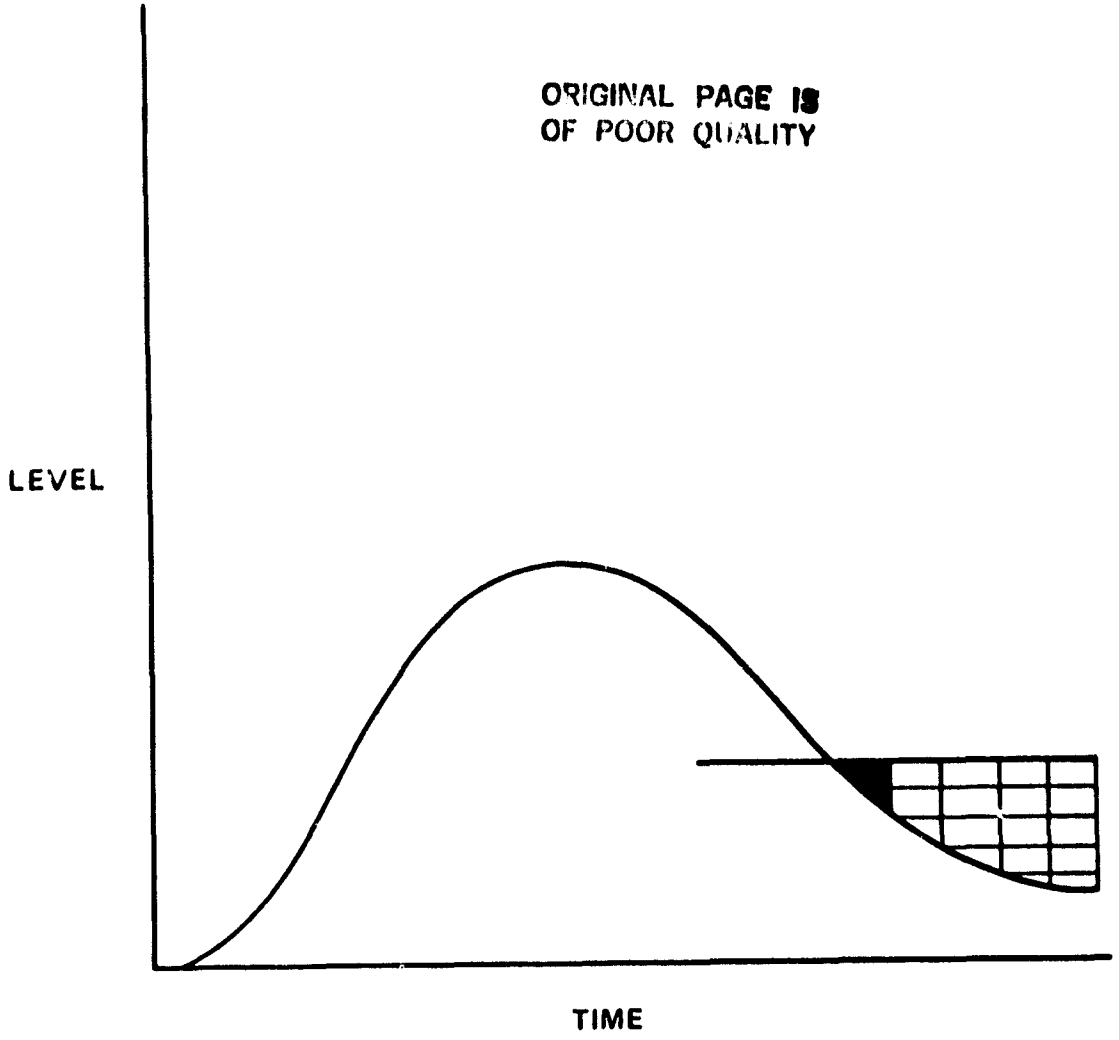


FIGURE 6. USING THE MANPOWER TO PERFORM NEW WORK

VALIDATION OF THE PROCESS

This methodology can be validated only by using the process and comparing the result to actual data. Since the maintenance phase has not yet occurred, a comparison of the method to an independently derived projection is an alternate approach. Figure 7 represents the use of the methodology on the Onboard Shuttle Software project. The figure presents the Rayleigh curve representing Release 19 of the flight software. Actual data from the project was compared with the curve as shown from 1/78 through 9/79. The results compared within 7% of real costs. The projected costs beyond 9/79 compared within 5% of projected costs derived by a bottom up estimate. The data from 1/77 to 1/78 were not comparable due to previous project costs embedded in the actual costs and functional design costs not included in the Rayleigh curve.

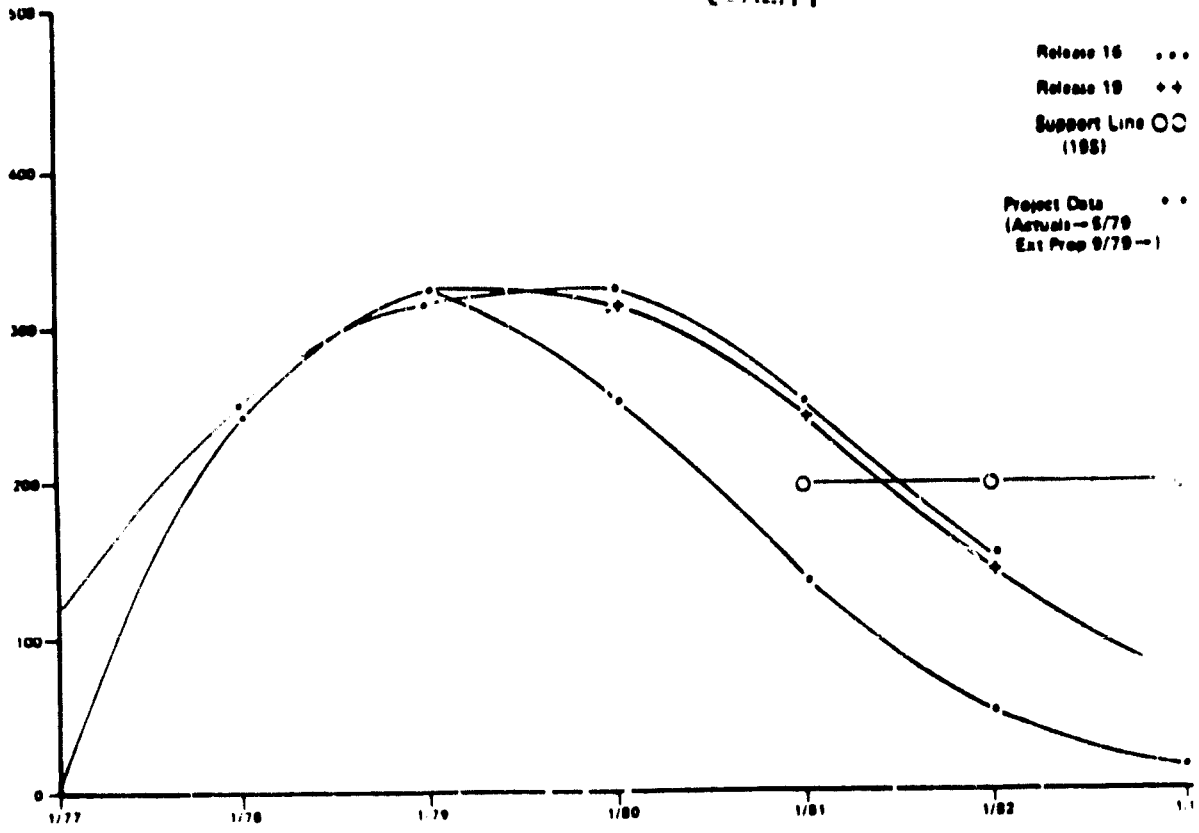


FIGURE 7. USING THE METHODOLOGY ON THE ONBOARD SHUTTLE SOFTWARE PROJECT

SUMMARY

The Maintenance Estimation Methodology is a method of projecting maintenance needs and required staffing levels. The methodology is summarized in the following steps:

SOFTWARE DEVELOPMENT AND MAINTENANCE PROJECTION

1. Use previous projection or actual data and assume that the work stops after last designated release.

2. Use Rayleigh curve method to project maintenance needs after the release.

3. Use matrix method to determine support line needed in a steady state period.

4. Compute the area between the two curves by integration.

5. Estimate the new work to be performed by transforming direct work estimates into project estimates.

6. Determine if new work fits under the support line. If not, either adjust schedules or phasing to reach support line.

7. Add new work scope and recompute Rayleigh curve to compare phasing and for basis of next projection.

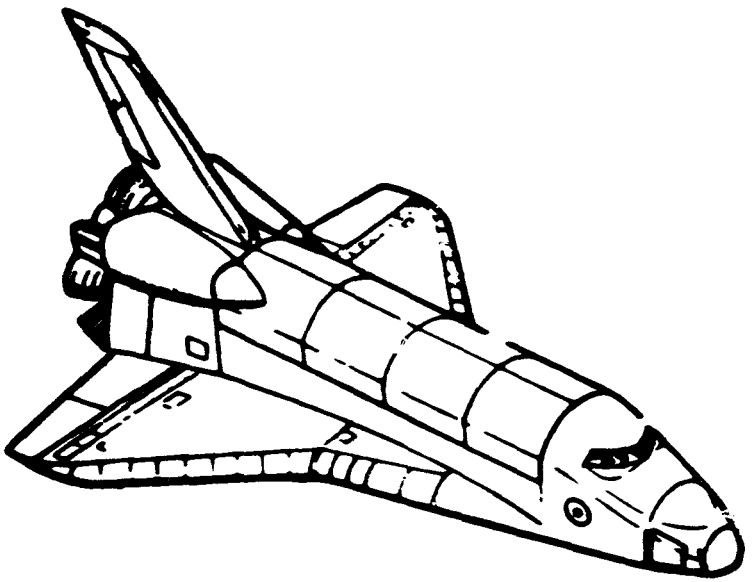
REFERENCES

1. Norden, Peter V., "Useful Tools for Project Management," Management of Production, M. K. Starr (Editor), Penguin Books, Inc., Baltimore, MD, 1970, pp. 71-101.

THE VIEWGRAPH MATERIALS

for the

K. RONE PRESENTATION FOLLOW



MAINTENANCE ESTIMATION METHODOLOGY

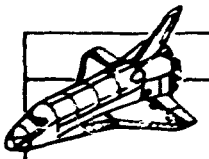
PRESENTATION

space shuttle programs

IBM
Federal Systems Division
1322 Space Park Drive, Houston 77058

K.Y. RONE

AUGUST 7, 1980



Title MAINTENANCE ESTIMATION METHODOLOGY | Date 1/16/80 | Page 1 of 9

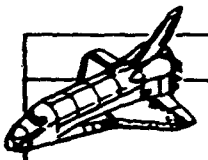
IBM

PREMISE

A SYSTEMATIC METHOD OF MAINTENANCE ESTIMATION IS NEEDED ON THE PROJECT

- CONFIDENCE OF BEING ABLE TO MEET REQUIREMENTS OF THE JOB IN THE FUTURE
- RATIONAL, SIMPLIFIED METHOD OF PROJECT ESTIMATION FOR RUNOUTS AND PROPOSALS
- REASONABLE ALLOCATION OF BLOCK UPDATE MANPOWER IN THE FUTURE.

953-1471

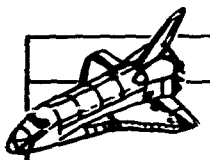


SPACE SHUTTLE PROGRAMS

Title MAINTENANCE ESTIMATION METHODOLOGY	Date 2/26/80	Page 2 of 9

IBM

WHAT IS NEEDED?

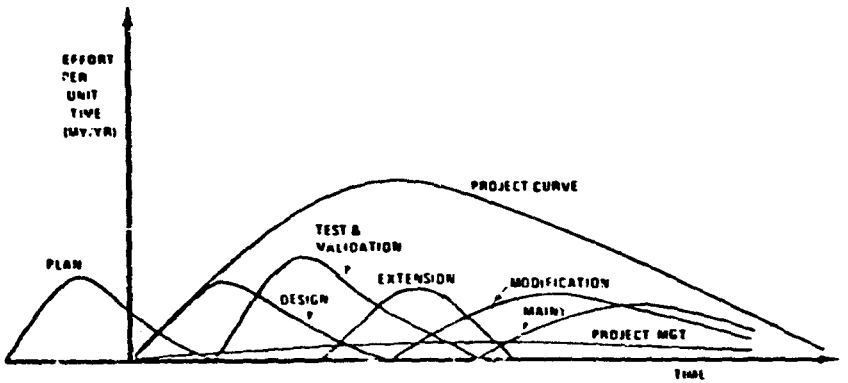


SPACE SHUTTLE PROGRAMS

Title	MAINTENANCE ESTIMATION METHODOLOGY	Date	1/16/80	Page 3 of 9

IBM

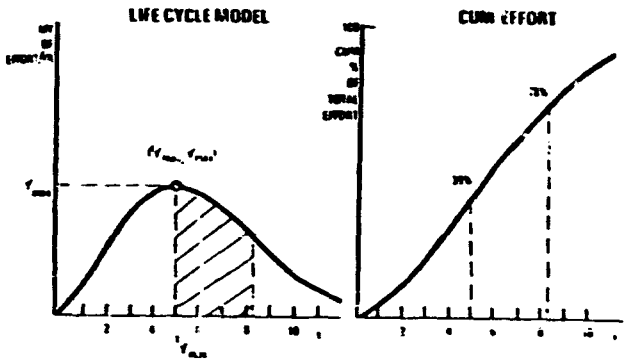
- NEED: A METHOD OF DETERMINING MAINTENANCE NEEDS IN THE FUTURE
- SOLUTION: RAYLEIGH CURVE METHOD
- USE:



953-1471

IBM

USE: (CONTINUED)



$Y' = 2Kate^{at^2}$ $\qquad Y = K(1-e^{at^2})$

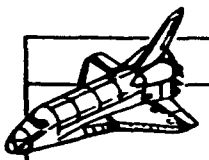
IMPORTANT PARAMETERS

K= TOTAL MY FOR ENTIRE PROJECT

$K = e^{\frac{1}{2}} \cdot Y'_{max} \cdot {}^1Y'_{max} = \sqrt{e} \cdot Y'_{max} \cdot {}^1Y'_{max}$

$a = 1/2\ t^2_{Y'_{max}}$

953-1471

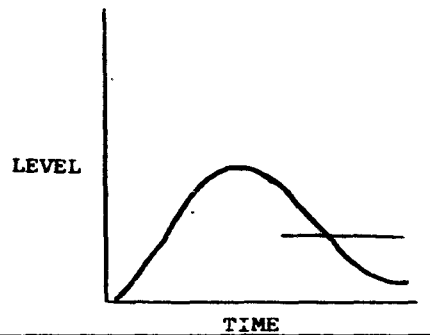


SPACE SHUTTLE PROGRAMS

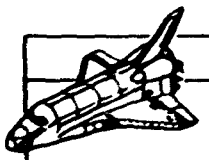
Title: MAINTENANCE ESTIMATION METHODOLOGY | Date: 1/16/80 | Page 5 of 9

IBM

- NEED: A METHOD OF DETERMINING A REASONABLE LEVEL OF SOFTWARE DEVELOPMENT SUPPORT IN A STEADY STATE PERIOD
- SOLUTION: MATRIX METHOD
- USE:
  - DETERMINE FUNCTIONAL ELEMENTS OF PROJECT
  - QUANTIFY MAINTENANCE NEEDS BASED ON: LEVEL = FUNCTION SIZE/((PRODUCTIVITY) (COMPLEXITY) (FACTOR))
  - CONSIDER CRITICAL SKILLS, LEVEL 3 TEST, OPERATIONS SUPPORT AND MANAGEMENT AND SUPPORT
  - SUMMARIZE FOR PROJECT
  - PLOT WITH RAYLEIGH CURVE
  - RAYLEIGH CURVE REPRESENTS CURRENT EFFORT REQUIRED BASED ON PAST EFFORT
  - SUPPORT LINE REPRESENTS LINE TO TEND TOWARD AND REACH IN STEADY STATE



953-1471

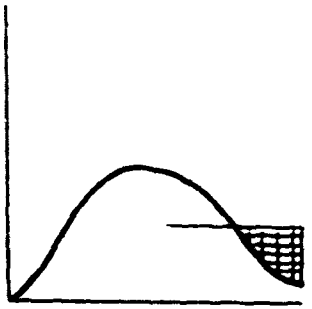


SPACE SHUTTLE PROGRAMS

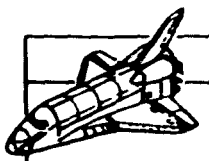
Title	MAINTENANCE ESTIMATION METHODOLOGY	Date 1/16/80	Page 6 of 9

IBM

- NEED: A METHOD OF DETERMINING MANPOWER AVAILABLE TO PERFORM NEW WORK
- SOLUTION: CALCULATE AREA BETWEEN CURVES
- USE:
  - INTEGRATE DIFFERENCE BETWEEN CURVES
  - EVALUATE OVER TIME OF INTEREST
  - AREA REPRESENTS EFFORT NOT USED IN MAINTENANCE OF PAST WORK WHICH CAN BE APPLIED TO NEW TASKS



953-1471



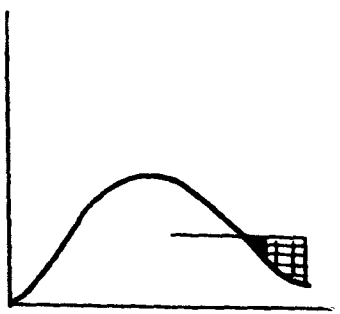
SPACE SHUTTLE PROGRAMS

Title	MAINTENANCE ESTIMATION METHODOLOGY	Date	1/16/80	Page 7 of 9

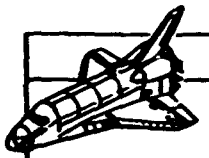
IBM

- NEED: A METHOD OF CONVERTING CR ESTIMATES TO TOTAL PROJECT COSTS
- SOLUTION: PROJECT COST EQUATIONS
- USE:
  - EXAMINE "CR" AND "FIXED" COSTS IN RECENT PROPOSALS
  - DETERMINE RELATIONSHIP BETWEEN CR AND TOTAL COSTS
  - PROJECT COST = 6.25 (CRA + CRS + CRV)

    WHERE CRA = APPLICATION CR COSTS
    CRS = SSW CR COSTS
    CRV = VERIFICATION CR COSTS
  - PROJECT COSTS REPRESENT THE COSTS WHICH WILL BE USED TO "FILL UP" THE AREA BETWEEN THE CURVES



953-1471

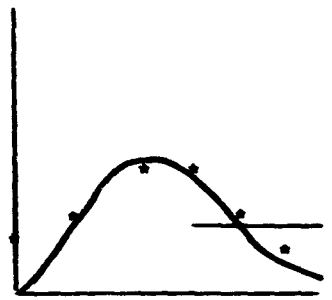


SPACE SHUTTLE PROGRAMS

Title MAINTENANCE ESTIMATION METHODOLOGY	Date 1/16/80	Page 8 of 9

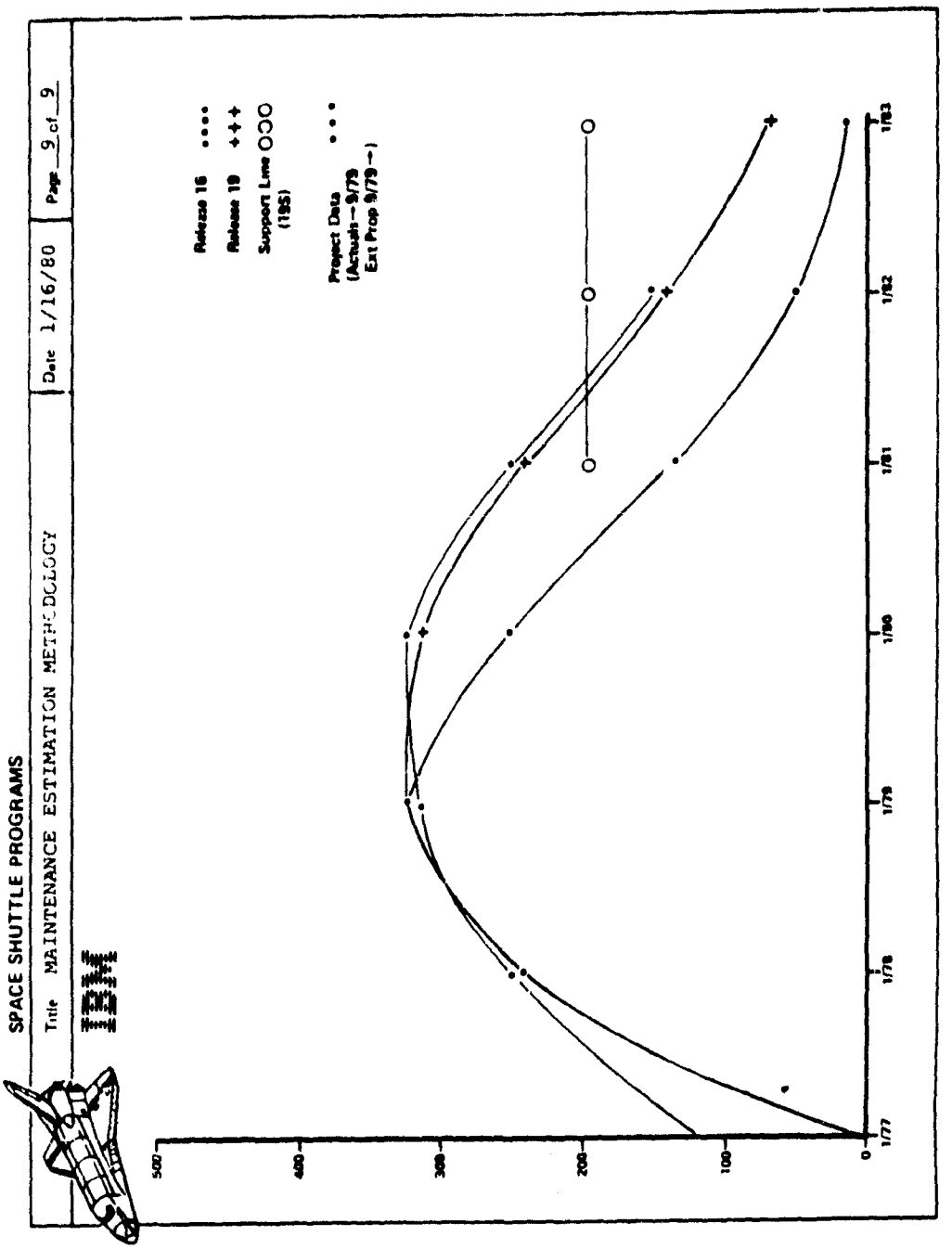
IBM

- NEED: VALIDATION
- SOLUTION: COMPARE RESULTS OF THE SCHEME TO PAST PROJECT DATA AND CURRENT PROJECTIONS
- COMPARISON:-
  - RESULTS COMPARED WITH RESULTS OF THE EXTENSION PROPOSAL
  - COMPARES WITHIN 7% OF REAL COSTS
  - COMPARES WITHIN 3-5% OF PROJECTED COSTS
  - EARLY COSTS NOT COMPARABLE DUE TO:
    - o ALT COSTS
    - o OFT FUNCTIONAL DESIGN COSTS NOT INCLUDED IN RAYLEIGH CURVE
  - COMPARISON FAVORABLE



953-1471

ORIGINAL PAGE IS
OF POOR QUALITY



STAFFING IMPLICATIONS OF SOFTWARE PRODUCTIVITY MODELS

Robert C. Tausworthe
Jet Propulsion Laboratory
California Institute of Technology
Pasadena, California

ABSTRACT

This paper investigates the attributes of software project staffing and productivity implied by equating the effects of two popular software models in a small neighborhood of a given effort-duration point. The first model, the "communications overhead" model, presupposes that organizational productivity decreases as a function of the project staff size, due to interfacing and intercommunication. The second, the so-called "software equation," relates the product size to effort and duration through a power-law tradeoff formula. The conclusions that may be reached by assuming that both of these describe project behavior, the former as a global phenomena and the latter as a localized effect in a small neighborhood of a given effort-duration point, are that (1) there is a calculable maximum effective staff level, which, if exceeded, reduces the project production rate, (2) there is a calculable maximum extent to which effort and time may be traded effectively, (3) it becomes ineffective in a practical sense to expend more than an additional 25-50% of resources in order to reduce delivery time, (4) the team production efficiency can be computed directly from the staff level, the slope of the intercommunication loss function, and the ratio of exponents in the software equation, (5) the ratio of staff size to maximum effective staff size is directly related to the ratio of the exponents in the software equation, and therefore to the rate at which effort and duration can be traded in the chosen neighborhood, and (6) the project intercommunication overhead can be determined from the staff level and software equation exponents, and vice versa. Several examples are given to illustrate and validate the results.

*The research reported in this paper was carried out at the Jet Propulsion Laboratory of the California Institute of Technology under a contract sponsored by the National Aeronautics and Space Administration.

STAFFING IMPLICATIONS OF SOFTWARE PRODUCTIVITY MODELS

Robert C. Tausworthe
Jet Propulsion Laboratory
California Institute of Technology
Pasadena, California

I. INTRODUCTION

Brooks [1], in The Mythical Man-Month proposed a simple model of software project intercommunication to show that, if each task of a large project were required to interface with every other task, then the associated intercommunication overhead would quickly negate the believed advantage of partitioning a large task into subtasks. While not meant to be an accurate portrayal of an actual project, the model effectively illustrated an increasing inefficiency symptomatic of projects too large to be performed by a single individual.

Putnam [2], in a 1977 study of software projects undertaken by the US Army Computer Systems Command, discovered a statistical relationship among product Lines of code, Work effort, and Time duration for those projects, whose best-fit formula was a power-law relationship, now referred to as the "software equation,"

$$L = c_k W^{0.33} T^{1.33}$$

(I have taken the liberty of changing Putnam's notation in order to be consistent with my notation in the remainder of the article.)

One rather startling extrapolation one may make from the software equation is that in order to halve the duration of any one of the projects studied, it would have taken 16 times the resources actually used! I say "extrapolation" because I suspect the software equation is more likely to be applicable incrementally—that is, if one were to require a 5% shortening of the schedule, then a 20% (actually 21.5%) increase in resources would be required.

In this paper, I will generalize both of these models parametrically, and suppose that both do describe the statistical trends of software projects in small neighborhoods about a chosen project situation. By equating the model behaviors in these neighborhoods, we shall be able to see how the parameters of one model relate to the parameters in the other. In addition, we shall discover some rather interesting facts about some actual projects for which published data exists.

II. A GENERALIZED INTERCOMMUNICATION OVERHEAD MODEL

Let us suppose that a software project is to develop L kilo-Lines of executable source language instructions, and that this number remains fixed over all our considerations of effort, duration, staffing, etc. That is, we shall suppose that the product size is invariant over the neighborhood of variability in these parameters—a project utilizing greater effort attempting to shorten the schedule slightly would produce the same program as a smaller effort requiring somewhat more time.

Let us denote by W the Work effort (in person-months) to be expended in the production of the L lines of code, and let the Time duration, in months, be denoted by T. Then the average full-time equivalent Staff size, S, in persons, is

$$S = W / T$$

and the overall team productivity can be defined as the number

$$P = L / W \qquad \text{(kilo-lines/person-month)}$$

Let us further suppose that the average fraction of time that each staff member spends in intercommunication overhead is dependent on the staff size alone, within a particular organizational structure and technology level, and let this fraction be denoted by t(S):

$$t(S) = \text{(intercommunication time) / (hours/mo. worked)}$$

Generally speaking, one intuitively expects t(S) to increase monotonically in S due to the expanding number of potential interfaces that arise as staff is increased.

But the individual average productivity of the staff, defined as the individual productivity during non-intercommunication periods, P_i, is somewhat greater than P, being related to it by

$$P = P_i \, [\, 1 - t(S) \,]$$

The relationship between the number of kilo-lines produced, the effort, and the staffing is

$$L = P_i \, W \, [\, 1 - t(S) \,]$$

Let us denote by W_0 and T_0 the effort and time, respectively, that would be required by a single unencumbered individual to perform the entire software task (assuming also that it could be done entirely by this individual, no matter how long it took). Then, with respect to the actual W and T, there is the relationship

$$W_0 = L / P_i = W [1 - t(S)] = T_0$$

This W_0 represents the least effort that must be expended, and T_0 is the maximum time that will be required. By substituting W/T for S, one obtains an effort-time tradeoff relationship

$$\omega = 1 / [1 - t(\omega/\tau)]$$

where $\omega = W/W_0$ and $\tau = T/T_0$ are "normalized" effort and duration, respectively.

The rate at which an increase in staffing results in an increase in normalized work effort is then

$$\frac{\partial \omega}{\partial S} = \omega^2 \, t'(S) > 0$$

where t'() refers to the derivative of t with respect to S. Because of the monotone character of t(S), an increase in staff leads to an increase in effort.

The overall staff production Rate, R, is the number of kilo-lines of code per month produced by the entire team of S persons,

$$R = P_i \, S [1 - t(S)]$$

The factor

$$\eta = [1 - t(S)]$$

is then the team production efficiency. Note that the normalized task effort is the inverse of the production efficiency,

$$\omega = 1 / \eta$$

The maximum rate of software production will occur when the derivative of R with respect to S becomes zero, a condition requiring a value S_0 that will satisfy the relationship

$$t'(S_0) = [1 - t(S_0)] / S_0$$

We shall refer to this staffing level as the maximum effective staff. Two particular examples of t(S) will serve to illustrate the characteristics of the intercommunication overhead model.

Linear Intercommunication Overhead. Let us assume first, as did Brooks, that the overhead is linear in staff,

$$t(S) = t_0(S-1)$$

that is, there is no overhead for 1 person working alone, but when there are S-1 other people, then each requires an average fraction t_0 of that individual's time. Under these assumptions, the maximum effective staff level is

$$S_0 = (1 + t_0) / (2\ t_0)$$

This value yields a maximum team production rate of

$$R_{max} = P_i\ S_0^2 / (2\ S_0 - 1)$$

and team efficiency

$$\eta_0 = (1 + t_0) / 2 = S_0 / (2\ S_0 - 1) \approx 0.5$$

This perhaps alarming result states that a team producing at its maximum rate is burning up half its effort in intercommunication overhead! The behavior is illustrated in Figure 1.

The normalized effort-duration tradeoff equation for this model takes the form

$$\tau = \frac{t_0\ \omega^2}{(1 + t_0)\ \omega - 1}$$

which has its minimum value at the maximum-production-rate point,

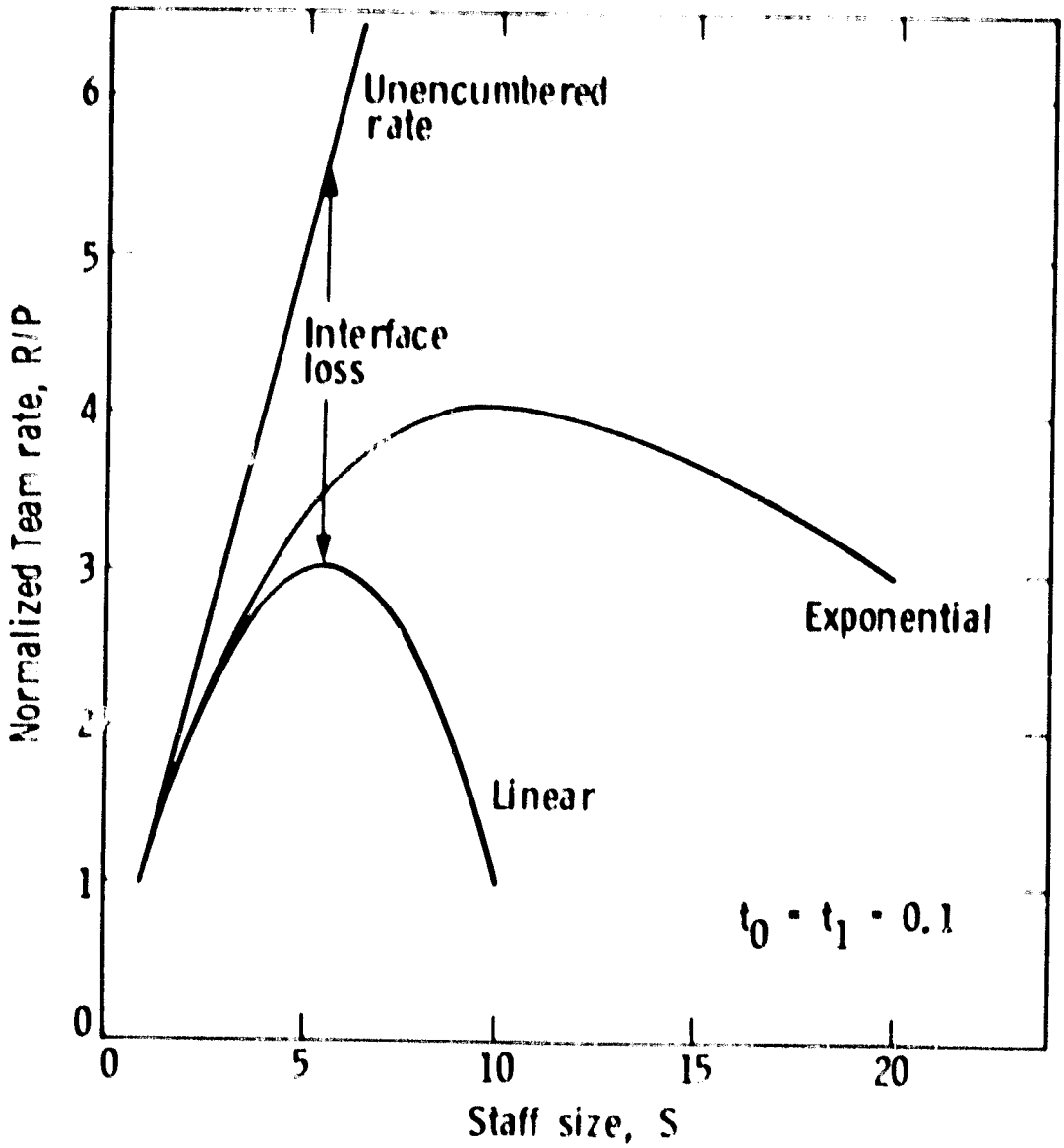
$$\tau_{min} = 4\ t_0 / (1 + t_0)^2 \approx 4\ t_0$$

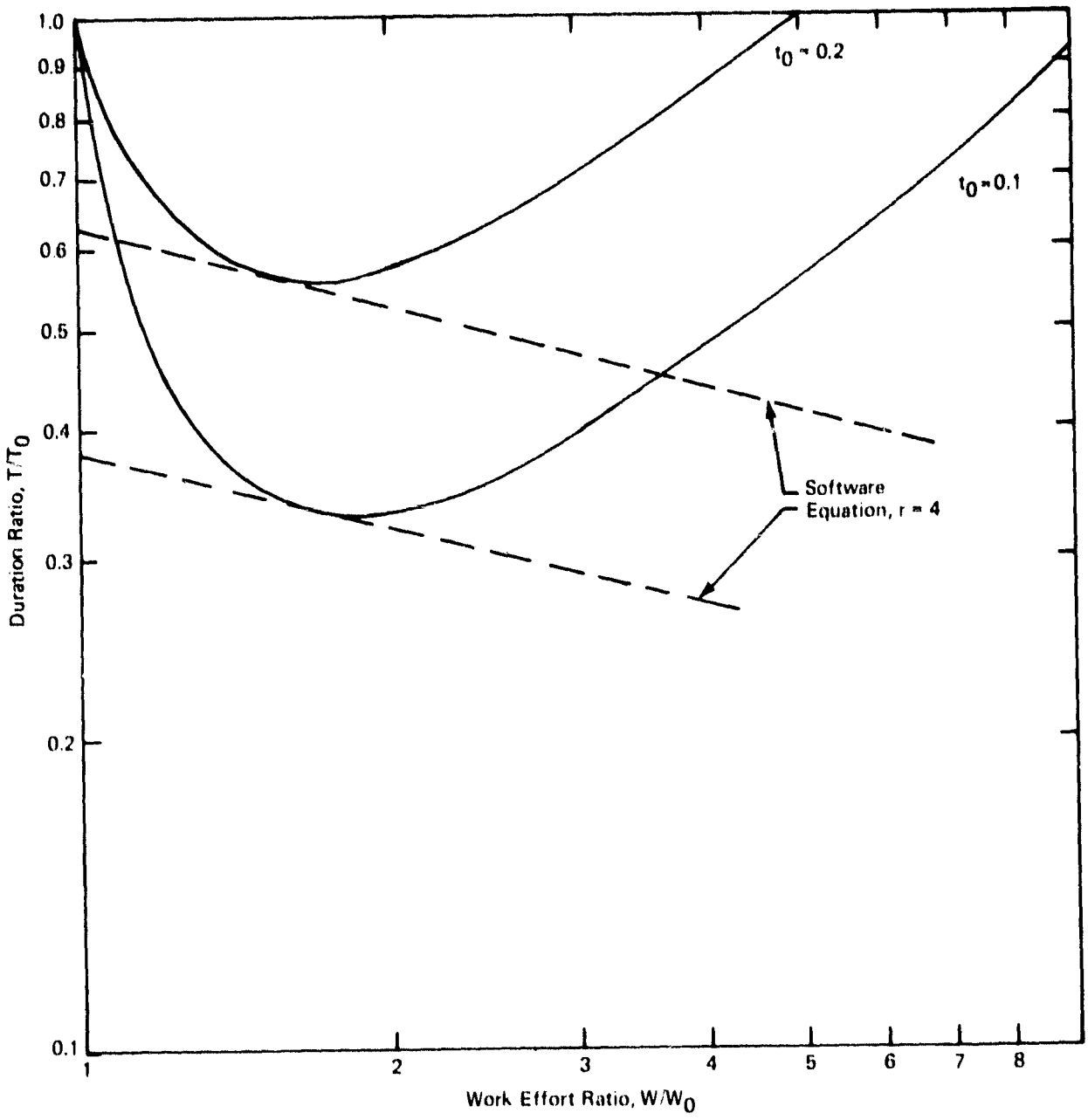
at which point the normalized effort is

$$\omega_0 = 2 / (1 + t_0) < 2$$

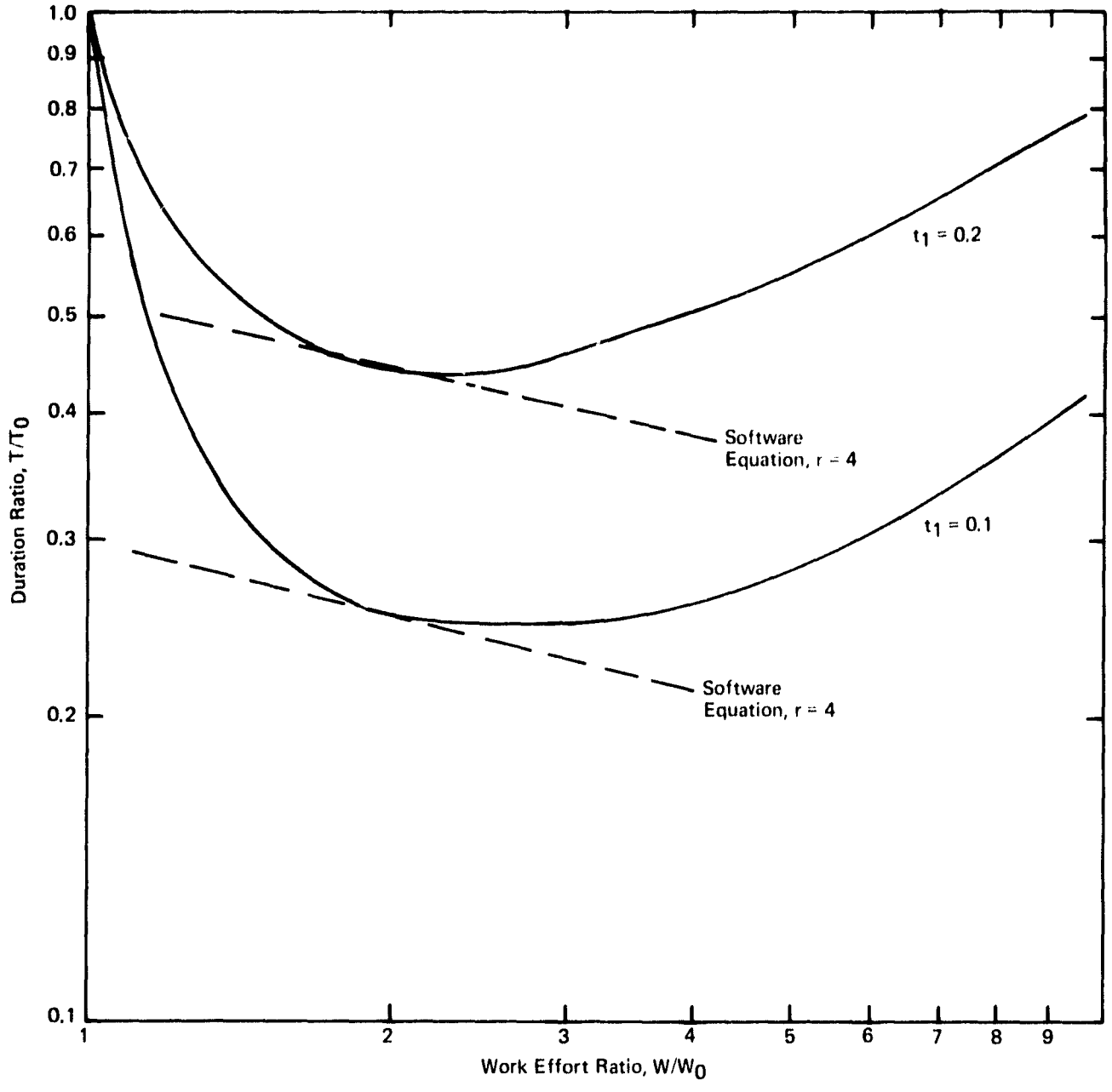
Figure 2 shows the characteristic of this tradeoff law at t_0 values of 0.1 and 0.2, for illustrative purposes.

According to this model, it <u>never</u> pays to expend more than twice the single-individual effort. Moreover, even though the ω producing the shortest schedule is less than 2, the effective range is much less than this, as shown in the figure. Effort can be traded for schedule time realistically only up to about 1.25 W_0, and a factor of two saving in time can only come about if the individual intercommunication can be kept below about 15% per interface.





ORIGINAL PAGE IS
OF POOR QUALITY



Exponentially Decaying Intercommunication Overhead. One unsettling aspect of the linear intercommunication overhead model is that, at some staffing level, the production rate goes to zero, and beyond, unrealistically into negative values. Perhaps a more realistic model is one which assumes that t(S) tapers off, never exceeding unity, at a rate proportional to the remaining fraction of time available for intercommunication as staff increases, or

$$t'(S) = t_1 \; [\; 1 - t(S) \;]$$

Then we are led to the form

$$t(S) = 1 - \exp[\; -t_1 (S - 1) \;]$$

The maximum effective staff in this case becomes

$$S_0 = 1 \; / \; t_1$$

and the maximum production rate is

$$R_{max} = P_i \; S \exp[\; -1 + 1/S \;] \approx P_i \; S \; / \; e$$

The team efficiency at this rate is

$$\eta_0 = \exp[\; -1 + 1/S \;] \approx 1/e$$

Now this is perhaps even more alarming a revelation than before, because it says that when producing software at the maximum team rate, that team is burning up 63% of its time in intercommunication! The consolation, as shown in Figure 1, is that the team performance under this assumed model is superior to that of the linear-time team model. More staff can be applied before the maximum effective staff level is reached.

The effort-duration tradeoff equation according to this model is

$$\tau = t_1 \; \omega \; / \; [\; t_1 + \ln(\omega) \;]$$

The minimum τ occurs at

$$\omega_0 = \exp(\; 1 - t_1 \;) < e$$

and the minimum value is

$$\tau_{min} = t_1 \exp(\; 1 - t_1 \;) \approx e \; t_1$$

The form of this tradeoff is shown in Figure 3 for t_1 values of 0.1 and 0.2, for illustrative purposes. Note that the minimum τ is much broader in this model, so that, although the actual minimum occurs when ω is about e in value, the realistic

effective range for ω is less than about 1.5. That is, it is not cost-effective to expend more than about 1.5 times the single-individual effort W_0 in an attempt to reduce the schedule time. A reduction in schedule by a factor of two is possible only when the individual intercommunication factor t_1 can be kept below 0.2.

Conclusions from Intercommunication Overhead Models. Both of the examples of intercommunication overhead above bespeak a maximum effective staffing level at which the project is 37-50% efficient. Beyond this point, further staffing is counter-productive. Both examples conclude that the maximum practical extent to which added effort is effective in buying schedule time is limited to about 25-50%. Significant schedule reduction factors are possible only when the intercommunication factors can be kept below 15-20%.

III. MATCHING THE SOFTWARE EQUATION MODEL

Let us generalize the Putnam Software Equation as the form

$$L = c_k W^p T^q$$

and let us define $r = q/p$, the exponent ratio. As in the previous section, L is held constant with respect to effort-duration tradeoff considerations. The value of p is assuredly positive: it generally requires more work at a given T to increase L. If q is positive, effort can be traded to decrease the schedule time required to deliver a given L. The larger r is, the larger the increase in effort required to shorten the schedule, and the larger the team production inefficiency. If q is zero, then L is a function of W alone, T is determined solely by the staffing level, T=W/S, and no additional effort is required to reduce schedule time (in the neighborhood in which the p and q=0 are valid). If q were ever to be negative, then an increase in W would render an increase in T, a situation indicating overmanned projects.

Substitution of $T = W/S$, differentiation with respect to S, and normalization of the software equation produces the result

$$\frac{\partial \omega}{\partial S} = \omega r / [S (1 + r)] = \tau r / (1 + r)$$

Let us now suppose that both the software equation and the intercommunications overhead model agree at the point (L, W, T). The two models can be equated by suitable choices of the "technology constant," c_k, and individual productivity, P_i. Then, in addition, let us suppose that the derivatives of effort with respect to staff level for both models also agree at this

point. Such can only be attempted when r > 0, because the derivative in the intercommunication overhead model is always positive. When this is the case, the two models may be said to agree in the neighborhood of the point (L, W, T).

Thus, by equating the derivatives, we arrive at a relationship between the parameters of the two models:

$$\frac{S\, t'(S)}{[\,1 - t(S)\,]} = \frac{r}{1 + r}$$

or

$$\eta = S\, t'(S)\, (\, r + 1\,) \,/\, r$$

Let us now examine this relationship for the two examples of the interface overhead model:

Linear Intercommunication Overhead. Substitution of the linear t(S) form into the neighborhood agreement condition yields

$$S = \left[\frac{2r}{1 + 2r} \right] \left[\frac{1 + t_0}{2t_0} \right] = S_0\, r \,/\, (\, r + 0.5\,)\,]$$

This equation states that the staffing level is related to the maximum effective staff point through the software exponent ratio, r. At the Putnam value, r = 4, the staffing level is 89% of the maximum effective level, and the team efficiency is

$$\eta = 0.55\, (\, 1 + t_0\,) \approx 55\text{-}65\%$$

$$\omega = 1.8 \,/\, (\, 1 + t_0\,) \approx 1.5\text{-}1.8$$

As seen in Figure 2, projects having this high an ω are at the point that extra effort is very ineffective.

Exponentially Decaying Intercommunication Overhead. By substituting the exponential form for t(S) into the neighborhood agreement condition, we find

$$S = r \,/\, [\, t_1\, (\, 1 + r\,)\,] = S_0\, r \,/\, (\, 1 + r\,)$$

Again, we see that the staffing level is related to the maximum effective staff via the exponent ratio. The Putnam value r = 4 produces

$$S = 0.8\ S_0$$

$$\eta = \exp[\ -(S-1)/S_0\ ] = \exp[\ -0.8 + t_1\ ] \approx 45\% - 55\%$$

$$\omega = 1/\eta = \exp[\ 0.8 - t_1\ ] \approx 1.8 - 2.2$$

Although this example indicates a somewhat more comfortable margin below maximum effective staffing than did the linear model, it nevertheless shows an alarmingly low cost inefficiency.

IV. EXAMPLES USING AVAILABLE DATA

Several data sets of project resource statistics published in the literature readily show that Putnam's value of r=4 is not universal. Specifically, Freburger and Basili [3] publish data which yield the following 3-parameter best power-law fits:

$$L_0 = 1.24\ W^{0.95}\ T^{-0.094} \qquad (\ r = -0.1\ )$$

$$L_1 = 0.22\ W^{0.78}\ T^{0.78} \qquad (\ r = 1.0\ )$$

in which L_0 is kilo-lines of delivered code, and L_1 is <u>developed</u> delivered code. It is interesting here to note that the former relationship is nearly independent of T, whereas the latter shows a definite beneficial W-T tradeoff characteristic. The negative q in the former relationship indicates that, on a <u>delivered code basis</u>, added resources in one of the projects would have extended the schedule! An equivalence between the software equation and the intercommunication overhead model cannot be established when r is zero or negative.

This data set is not the only one to show a negative q: Boehm [4], in his <u>Software Economics</u> book, has a data base used to calibrate his COCOMO software cost model. A 3-parameter best power-law fit to the adjusted data produces the relationship

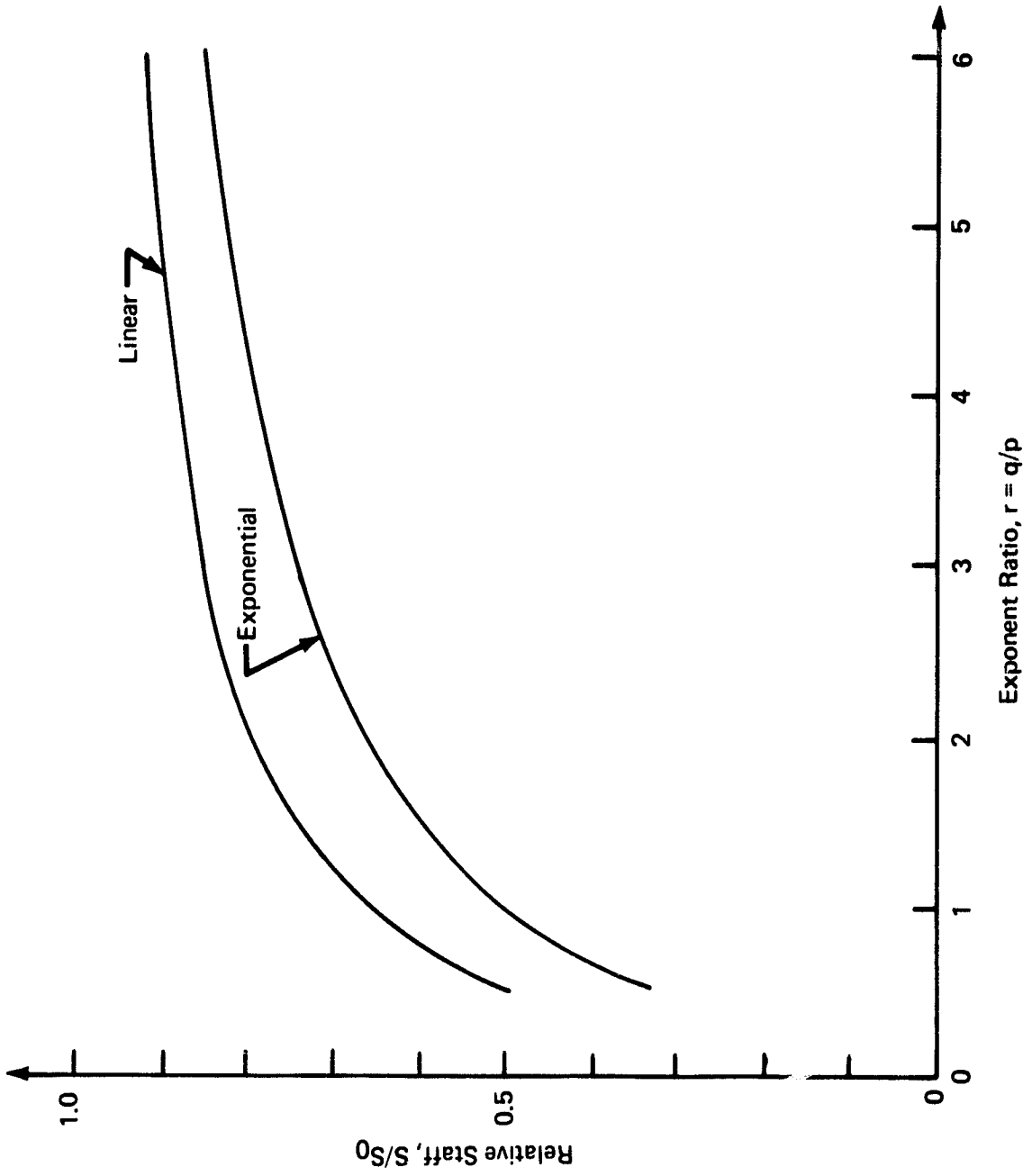
$$L = 0.942\ W^{0.675}\ T^{-0.028} \qquad (\ r = -0.41\ )$$

Again, the tradeoff equation indicates that the projects in that data base were perhaps overmanned.

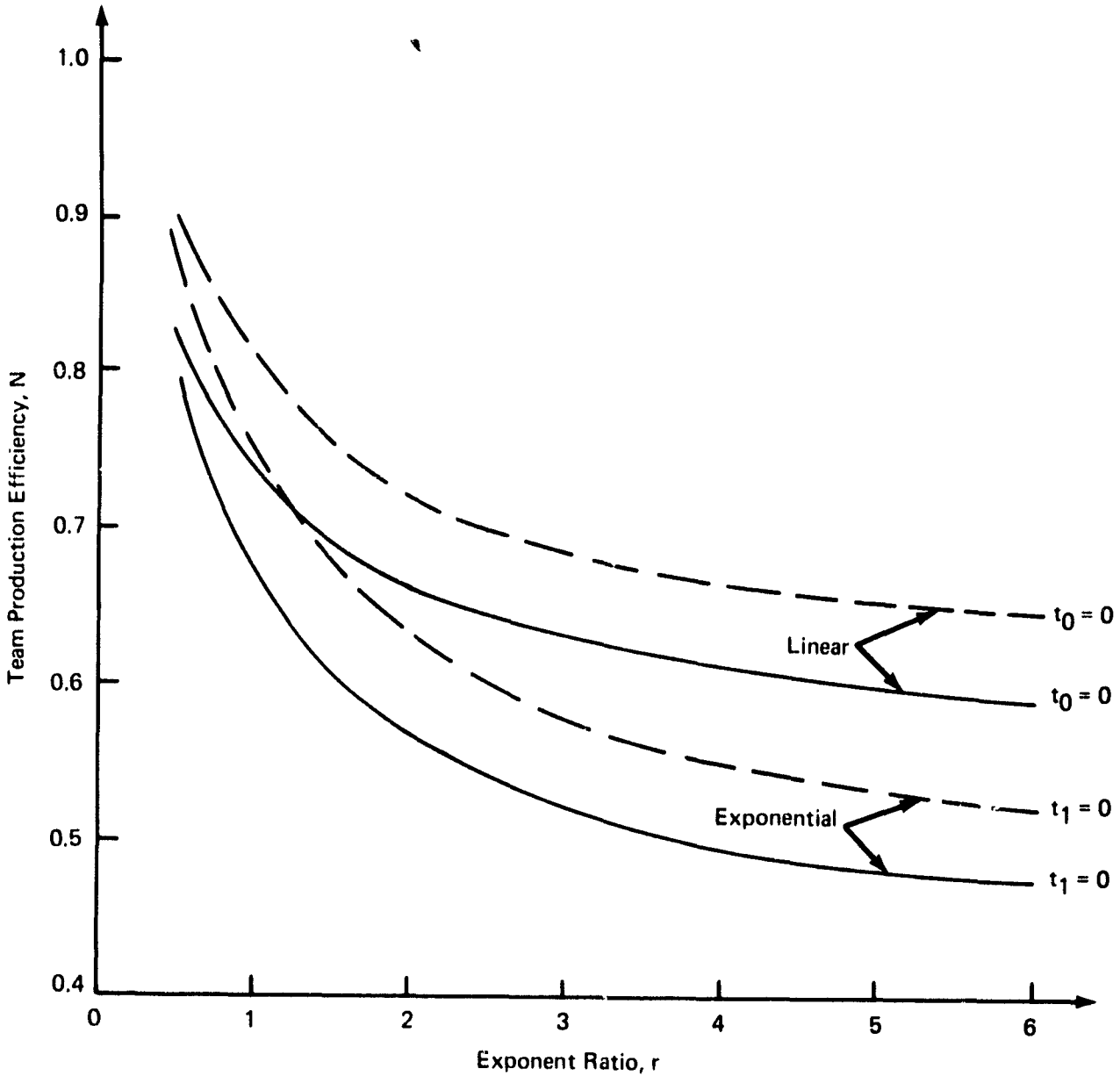
Gaffney [5], on the other hand, did a 3-parameter best power-law fit of IBM data (Federal Systems Division, Manassas) to arrive at the relationship

$$L = c_k\ W^{0.63}\ T^{0.56} \qquad (\ r = 0.88\ )$$

This last value of r aligns more closely with the Freburger-Basili value for developed delivered code.



ORIGINAL PAGE IS
OF POOR QUALITY



V. CONCLUSION

This article has shown that when there is a positive effort-duration tradeoff relationship in a software project, it is possible to estimate the team production efficiency and proximity to maximum effective staffing. These figures can be used to advantage by software managers who must judge the effectiveness of increasing resources in order to shorten schedules. It points out the necessity of keeping accurate records of software project statistics, so that the parameters in the model can be estimated accurately.

Low values of r in an organization are a mark to be proud of, showing efficiency in terms of structuring subtasks for clean interfaces. High (or negative) values of r may be indicative of overall task complexity, volatility of requirements, organizational inefficiency, or any number of other traits that tend to hinder progress. The value of r may thus be treated as a figure of merit--a measurable statistic indicative of the efficiency of a set of projects in performance of assigned tasks.

The ratio S/S_0 is another indicator for management. When low, it indicates that adding resources can potentially help a project in trouble. If closer to unity, it is a warning that adding resources may not help, will not appreciably shorten the schedule, will incur expense at a low return in productivity, and, if applied often in other projects, will thereby contribute to an organizational reputation for expensive software.

REFERENCES

1. Brooks, F. P., The Mythical Man-Month, Addison-Wesley Pub. Co., Reading, MA, 1975.

2. Putnam, L. H., "Progress in modeling the software life cycle in a phenomenological way to obtain engineering quality estimates and dynamic control of the process," Second Software Life Cycle Management Workshop, sponsored by US Army Computer Systems Command and IEEE Computer Society, Atlanta, GA, Aug. 1978.

3. Freburger, K., and Basili, V. R., "The Software Engineering Laboratory, Relationship Equations," Report TR-764, University of Maryland Computer Science Center, College Park, MD, May, 1979.

4. Boehm, B. W., Software Economics, Prentice-Hall Publishing Co., Englewood Cliffs, NJ, 1982.

5. Gaffney, J. E., "An Approach to Software Cost and Schedule Estimation," submitted to Journal of Defense Systems Acquisition Management, (pending).

THE VIEWGRAPH MATERIALS

for the

R. TAUSWORTHE PRESENTATION FOLLOW

STAFFING IMPLICATIONS OF SOFTWARE PRODUCTIVITY MODELS

Robert C. Tausworthe

- INTERCOMMUNICATIONS OVERHEAD MODELS
- PUTNAM SOFTWARE EQUATION
- COMBINED EFFECTS
- CONCLUSIONS

NOMENCLATURE

L = LINES OF DELIVERED SOURCE CODE (THOUSANDS)

W = WORK EFFORT (PERSON-MONTHS)

S = AVERAGE FULL-TIME EQUIVALENT STAFF (PERSONS)

P = PRODUCTIVITY (KILO-LINES OF CODE/PERSON-MONTH)

R = TEAM PRODUCTION RATE (KILO-LINES/MONTH)

INTERCOMMUNICATION OVERHEAD MODEL

$t(S)$ = (INTERCOMMUNICATION TIME)/(hrs/mo. WORKED)

$P = P_i [1 - t(S)]$

P_i = INDIVIDUAL PRODUCTIVITY DURING NON-INTERCOMMUNICATIONS

$L = P_i W [1 - t(S)]$

$R = P_i S [1 - t(S)]$

$t(S) = 0$ FOR $S \leqslant 1$

$t(S)$ INCREASES MONOTONICALLY FOR $S > 1$

EFFORT - DURATION TRADEOFF
INTERCOMMUNICATION OVERHEAD MODEL

$$\frac{W}{W_0} = \frac{1}{1 - t\left(\frac{W}{W_0} \cdot \frac{T_0}{T}\right)}$$

WHERE THE SINGLE-INDIVIDUAL-TASK W_0, T_0 VALUES ARE

$$W_0 = T_0 = L/P_i$$

W_0 IS LEAST EFFORT REQUIRED

T_0 IS LONGEST TIME REQUIRED

- LINEAR INTERCOMMUNICATION OVERHEAD

$$t(S) = t_0 (S-1) \quad \text{FOR} \quad S \geqslant 1$$

$$\frac{T_{min}}{T_0} = \frac{4t_0}{(1+t_0)^2} \approx 4t_0 \quad \text{AT} \quad \frac{W}{W_0} = \frac{2}{1+t_0} < 2$$

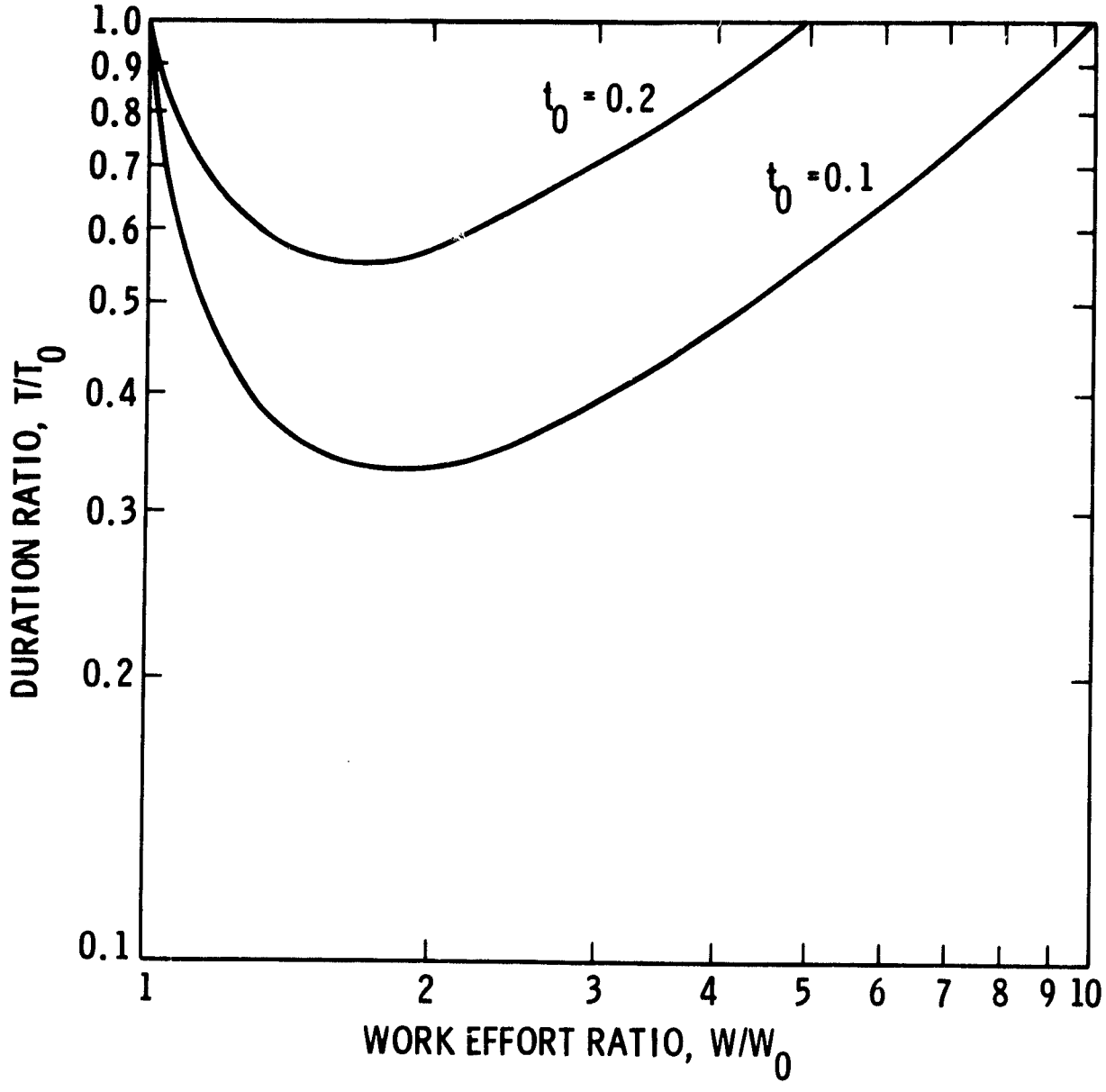
- EXPONENTIAL DELAY INTERCOMMUNICATIONS OVERHEAD

$$t(S) = 1 - \exp\,[-(S-1)t_1]$$

$$\frac{T_{min}}{T_0} = t_1 \exp\,(1-t_1) \quad \text{AT} \quad \frac{W}{W_0} = \exp\,[1-t_1] < e$$

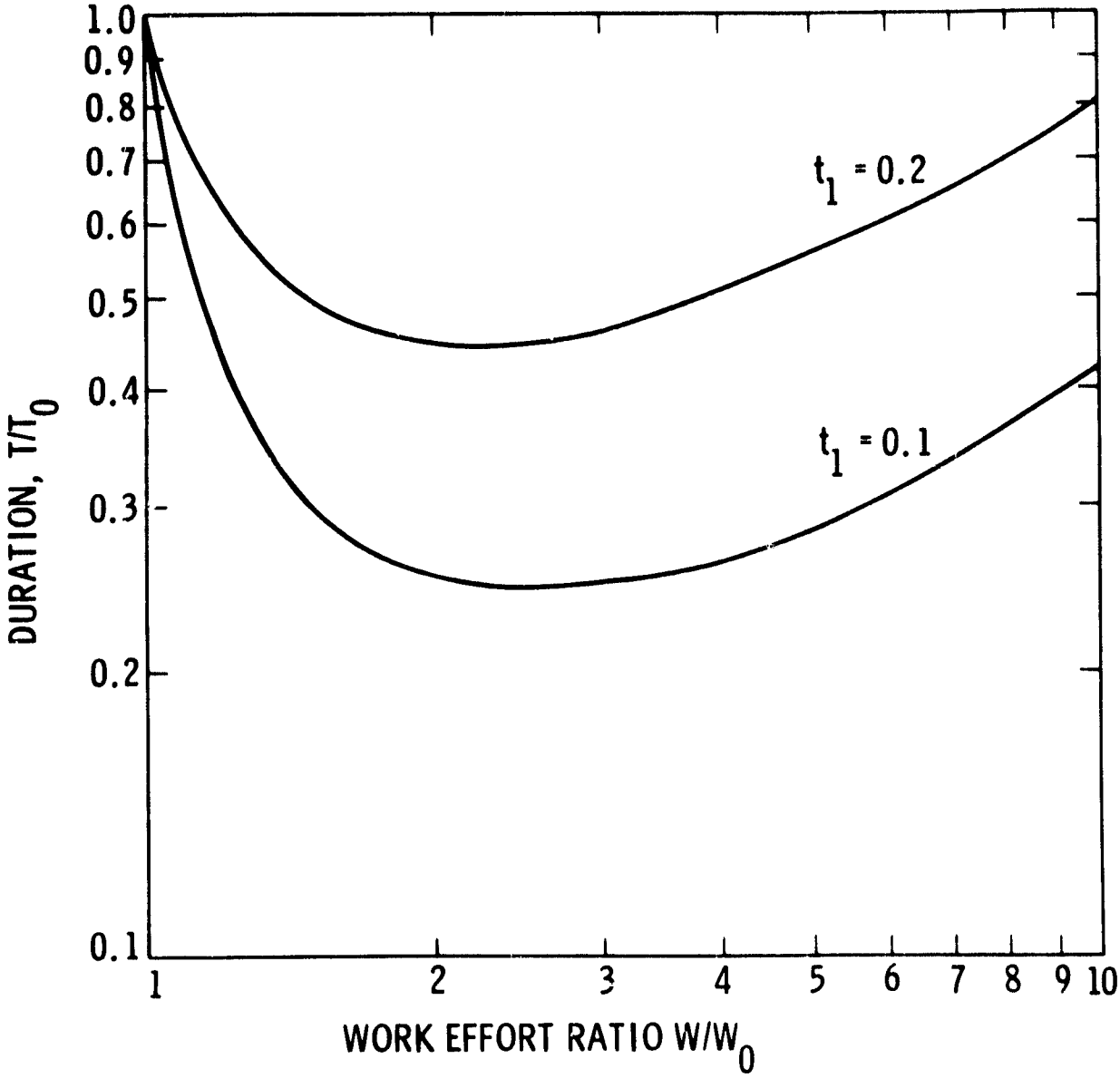
- S_0 = STAFF SIZE AT T_{min} IS THE "MAXIMUM EFFECTIVE STAFF"

TIME - EFFORT TRADEOFF
LINEAR OVERHEAD



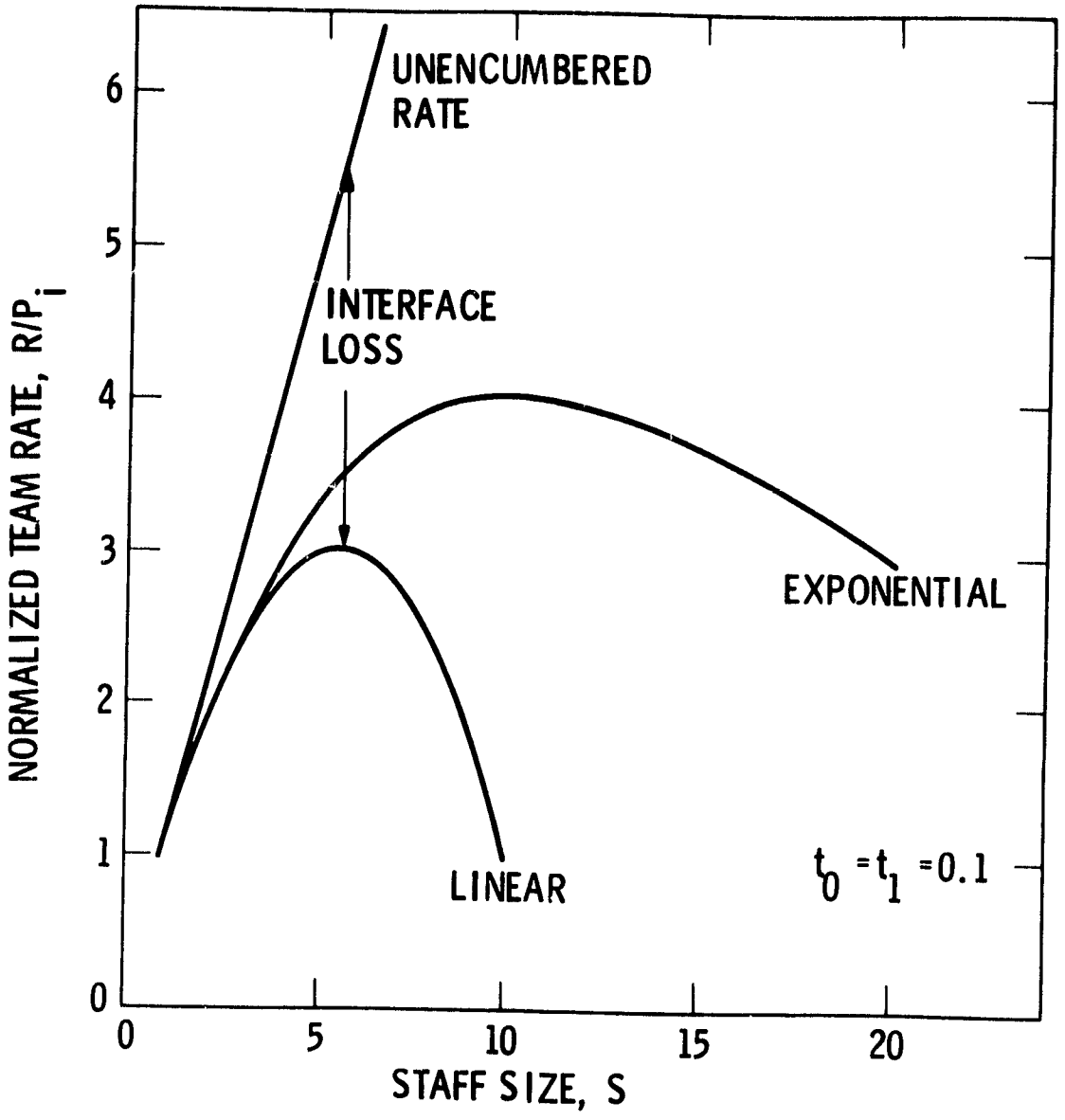
RCT-7
12-1-82

TIME - EFFORT TRADEOFF
EXPONENTIAL OVERHEAD



RCT-8
12-1-82

PRODUCTION RATE



RCT-9
12-1-82

SOFTWARE EQUATION

GENERAL FORM

$$L = c_k W^p T^q$$

- DENOTE $r = q/p$
- PUTNAM's ORIGINAL EVALUATION

$$L = c_k W^{0.33} T^{1.33}$$

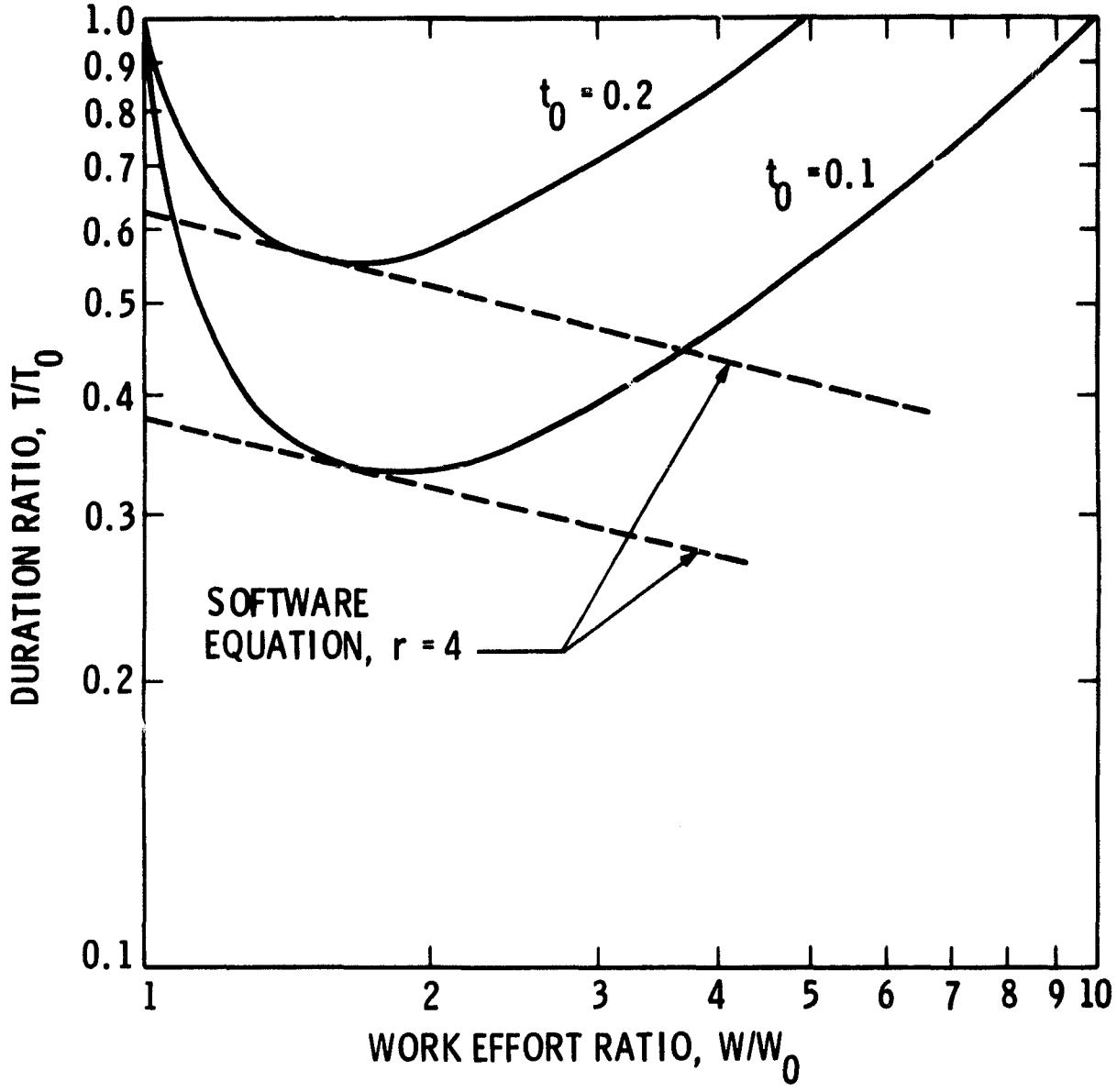
- DEFINES TIME-EFFORT TRADEOFF
- PUTNAM's VALUE OF $r = 4$

NEIGHBORHOOD EQUIVALENCING

- ASSUME OVERHEAD MODELS DESCRIBE GLOBAL EFFECTS OF STAFF SIZE ON PRODUCTIVITY FOR GIVEN L
- ASSUME SOFTWARE EQUATION EXPLAINS LOCALIZED BEHAVIOR IN NEIGHBORHOOD OF A PARTICULAR (W, T) POINT FOR GIVEN L
- MAKE BOTH MODELS AGREE AT (W, T) AND HAVE SAME SLOPE AT THIS POINT, FOR GIVEN L, BY PROPER CHOICE OF TECHNOLOGY CONSTANT, c_k, AND INDIVIDUAL PRODUCTIVITY, P_i
- NEIGHBORHOOD EQUIVALENCE CRITERION

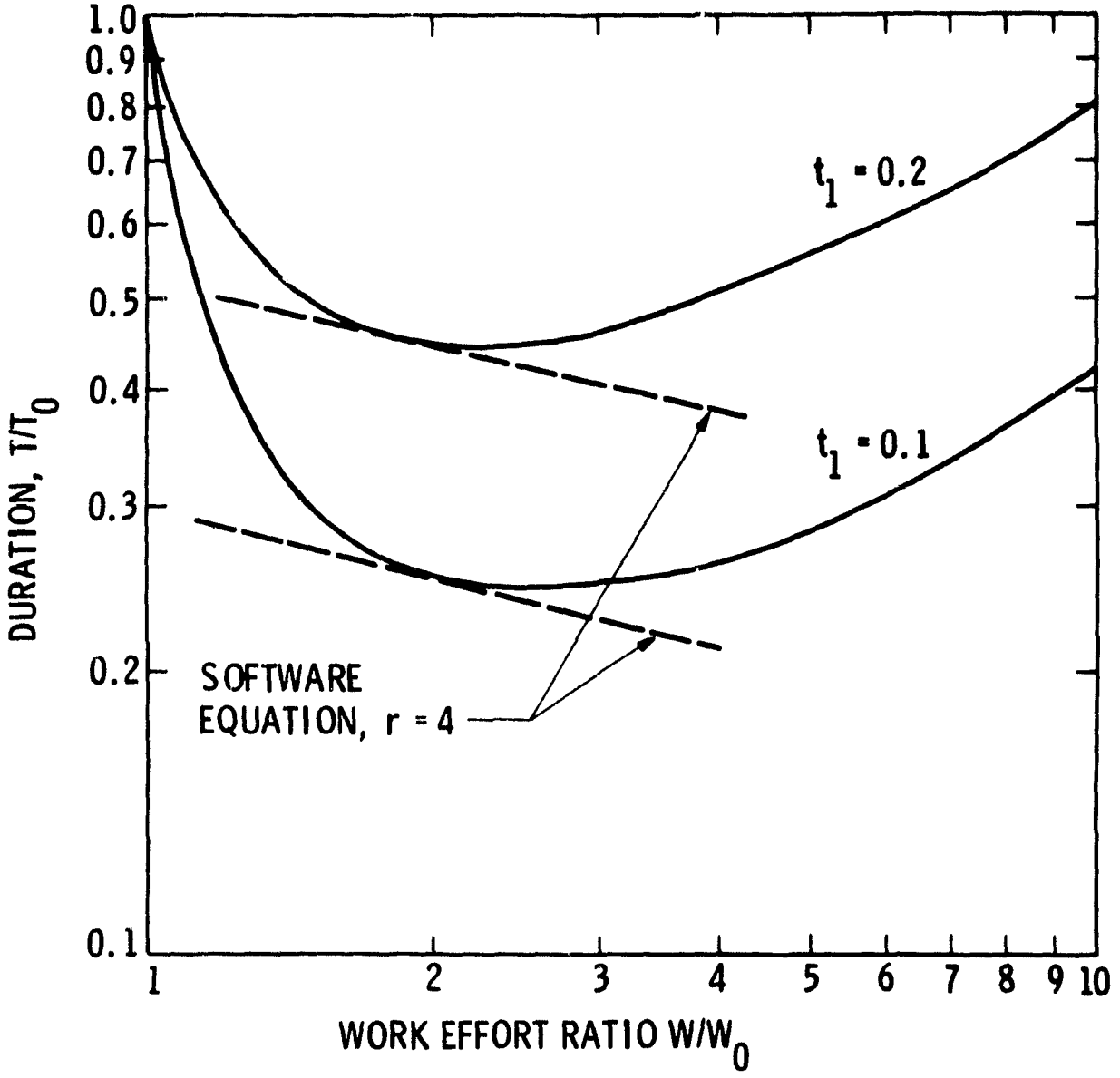
$$\frac{S\,t'(S)}{1 - t(S)} = \frac{r}{1 + r}$$

LOCAL BEHAVIOR, LINEAR OVERHEAD



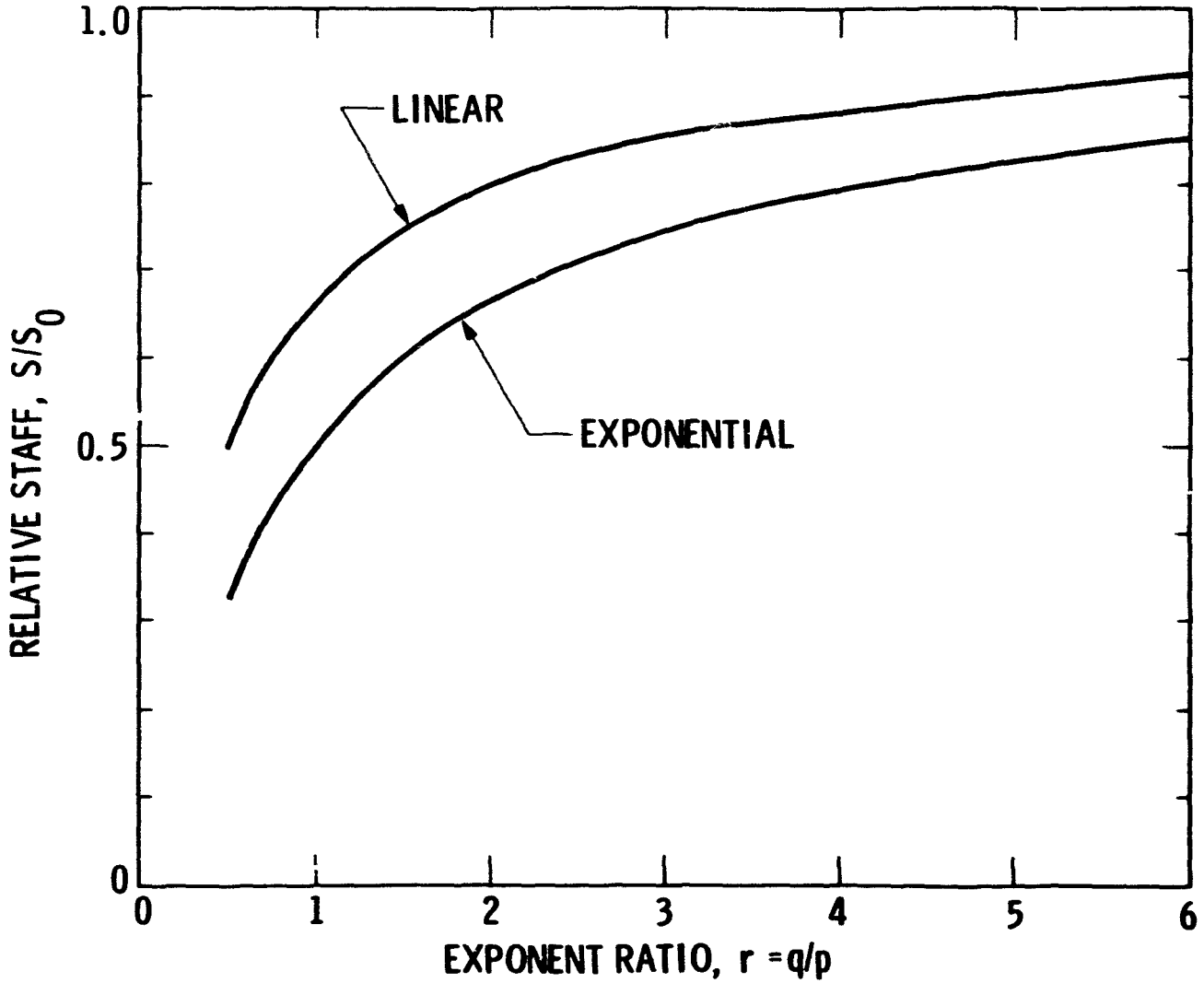
RCT-12
12-1-82

LOCAL BEHAVIOR, EXPONENTIAL OVERHEAD



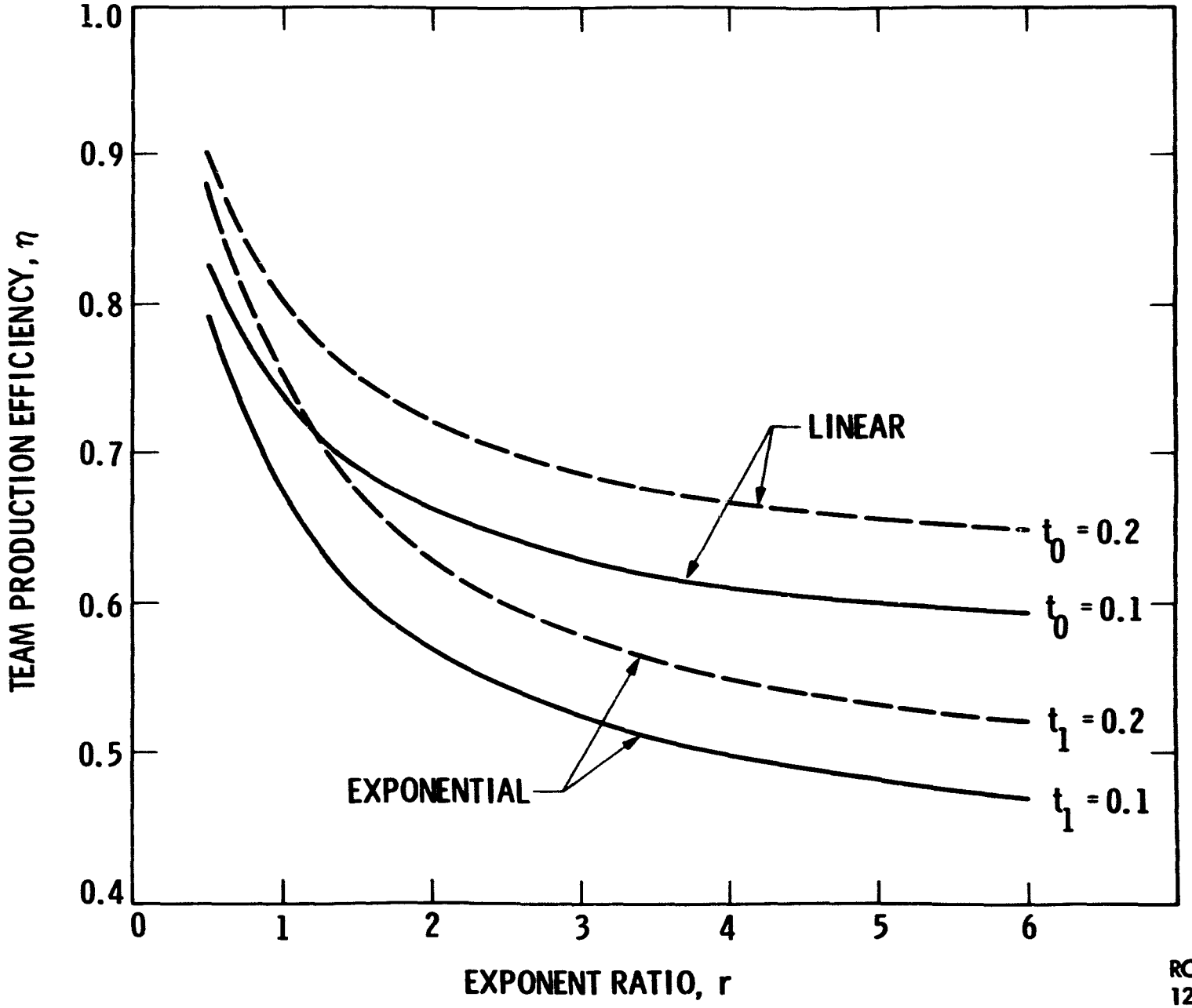
RCT-13
12-1-82

RELATIVE STAFF VS EXPONENT RATIO



RCT-14
12-1-82

PRODUCTION EFFICIENCY



RCT-15
12-i-82

EXPONENT RATIO DETERMINATIONS

- PUTNAM's ORIGINAL VALUE, r = 4

- FREBURGER-BASILI (U. OF MD)

  r = 1.0 (DEVELOPED, DELIVERED CODE)

  r = -0.1 (DELIVERED CODE)

- GAFFNEY (IBM-MANASSAS)

  r = 0.88

- BOEHM (TRW)

  r = -0.041 (ADJUSTED DATA)

  r = 0.086 (RAW DATA)

CONCLUSIONS

- TIME AND EFFORT CAN BE TRADED ONLY SO FAR
- THE EXPONENTS OF THE SOFTWARE EQUATION ARE RELATED TO THE S/S_0 RATIO, AND THEREFORE ARE INDICATORS OF HOW NEAR A PROJECT IS TO BEING OVERSTAFFED
- WHEN S/S_0 IS NEAR UNITY, ADDITIONAL STAFFING WILL NOT HELP A PROJECT
- IT IS NEVER EFFECTIVE TO APPLY MORE THAN TWICE THE SINGLE-INDIVIDUAL-EFFORT TO SHORTEN SCHEDULE TIME
- THERE IS A NEED FOR MORE STATISTICAL STUDY OF r AS A FUNCTION OF OTHER PROJECT CHARACTERISTICS

D13
12

Estimates of Software Size From State Machine Designs

Robert N. Britcher
IBM, Federal Systems Division
Gaithersburg, Md.

John E. Gaffney*
National Weather Service
Silver Spring, Md.

* On leave from IBM Corporation, Federal Systems Division

There is a greatly evident need for improving the estimates of the amount of function to be provided by a software system. State Machine models (1,2) are being employed to record software designs as they evolve. So, it appears natural to attempt to derive estimates of the amount of code that will ultimately result from these designs by using quantities directly available from them as they are created. This paper demonstrates that the length, or size (in number of Source Lines of Code) of programs represented as state machines can be reliably estimated in terms of the number of internal state machine variables. Variables, here, are defined as the unique data required by a state machine's transition function, not the data retained in the state machine's memory. They are equivalent to Halstead's (3) operands. Data collected from the SACDIN project (4) was used to develop software size estimating formulas for a software system from which the state machine representation is available at various levels of abstraction. Hence, the methodology presented should be employable at successive stages of the development process to provide estimates (with, hopefully) increasing accuracy.

An important aspect of developing software is the derivation of estimates of the amount of function (typically presented as a SLOC count) the system is to provide. This paper presents code size estimation formulas that can be successively applied as the design for a software system evolves. The estimation of software size and development cost (assuming certain rates) in terms of man months per thousand lines of code (see reference 5) can be made relatively early in design and refined as the design effort proceeds. The code size estimation formulas can be applied to a state machine conceptualization of a software system at the highest level and individual procedures at the lowest.

A program can be regarded, and hence estimated, evaluated, and/or compared with another program in a number of different ways. Here, we are concerned with two principal ways, the linguistic and the structural. From the linguistic point of view, a program can be regarded as a string of tokens or symbols. Halstead (3), who did pioneering work using the linguistic approach, demonstrated a fundamental relationship between the size of the operand and operator vocabulary and the length of the program text, stated in terms of the number of tokens or symbols constituting it. This relationship is:

$N = \eta_1 \log_2 \eta_1 + \eta_2 \log_2 \eta_2$, where N = number of tokens, η_1 = operator vocabulary size, and η_2 = operand vocabulary size.

In assembly code, the "operator" corresponds to the op. code symbol, and the "operand" corresponds to the "address" or operand field of the instruction. Also, "I", the number of instructions is proportional to "N", the number of tokens; or I = aN. In fact, $I = b.n_2 \log_2 n_2$, approximately, as shown by Gaffney for the case of AN-UY K-7 assembly code (9). Christensen et al. have also observed that "program size is determined by the data that must be processed by the program (10)". We assert that the "variable count", obtained from the state machine design, at the "procedure level" (as described more fully below) corresponds to "n_2", the operand vocabulary size in Halstead's

formulas. It is of interest to note that relationships similar to those developed by Halstead and others for software, part of the material that may be termed "software linguistics", have been noted between text length and vocabulary size in natural languages by Herdan (6).

From the structural point of view, a program can be considered principally in terms of data flow or in terms of function. In the former, the amount of function, stated in terms of the number of lines of code, is related to the data flow into and out of each module (see Kafura and Henry (7)) or into and out of a program as a whole (see Albrecht (8)). In the function approach, the number of unique inputs and outputs for a procedure, a module, or a program as a whole is implied by the size of the function in that software element. Whereas, here, we assert the equivalence between the Halstead approach and the function approach, by relating the number of variables in a state machine procedure to the number of source lines of code: the variables are equivalent to the operands in Halstead's formulas.

A program, or a subdivision of one, such as a module, can be represented as a "state machine", as depicted in Figure 1. The "State Machine" consists of two principal parts, the "transition function" and the "state data". The former gives rise to the actual code. The latter is the "memory" of the program. The transition function, call it "T" is a function whose elements are ordered pairs of ordered pairs (2), to wit:

$$T = [\,(\text{present state, input}), (\text{new state, output})\,]\,.$$

Thus, "T" really symbolizes the combinational logic of the program, not different in principle from a program without memory. The state machine characterization of a program is an adaptation of the "Mealy-Moore" model of sequential machines originally developed to represent automation in general and telephone switching circuitry in particular (11).

As described by Britcher and Moore (4), the SACDIN Dialog Manager was designed using the state machine model. Some 8000 lines of code (S/370 assembly plus some macros, including comments), were written, based on a state machine decomposition consisting of 20 machines, comprising 74 transitions, or procedures. We derived several formulas (by regression). One of them was:

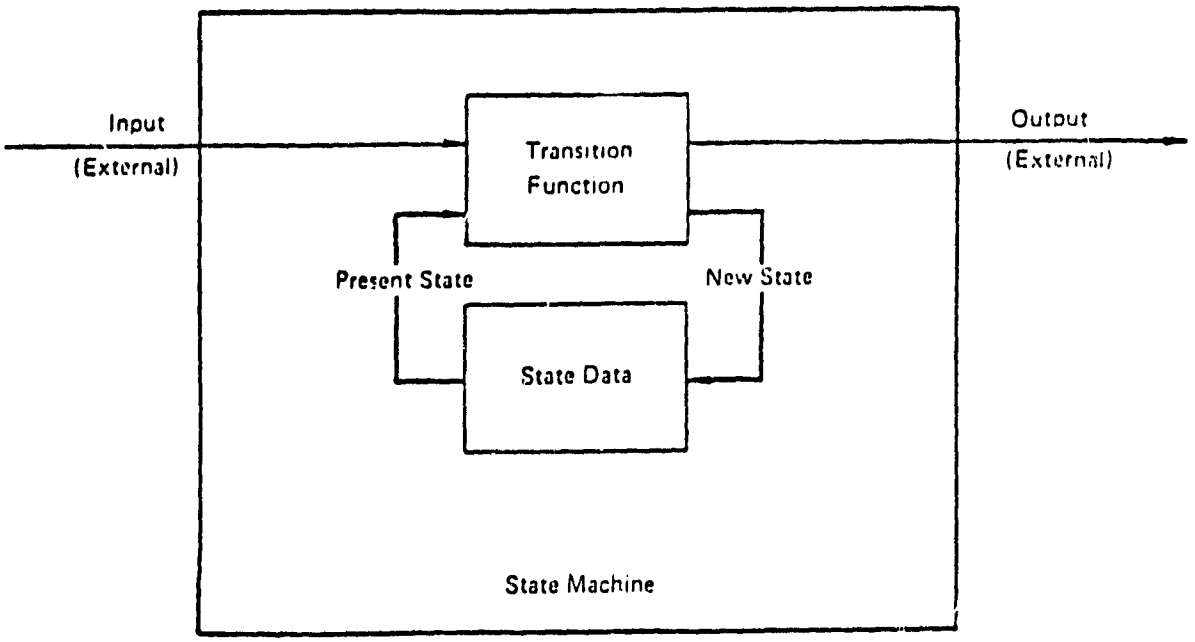
$S = 8.825 \times V\log_e V$, where S = estimated number of SLOC, including comments (about 40%).

(The statistics of the fit, to the data from which it was derived) is given in the table below:

Size Estimating Formula	Relative Error (1) $\frac{(\hat{S} - S)}{S}$ Avg. by Procedure	Std. Deviation by Procedure	Avg. Overall
$S = 8.825 \times V\log_e V$	.027	.564	-.0097
$S = 21.3282xV$	.222	.518	.0845

FIGURE 1

State Machine Representation of a Program



T = [(p. state, input); (n. state, output)]

Note: (1) S = estimated SLOC's (w/comments); S = actual SLOC's (w/comments)

The variable V is the "variable count" obtained from the state machine design. It corresponds to η_2, the number "operands" in Halstead's formulas.

The software code size formula, $S = 8.825 \times V \log_e V$, was verified using the data from another major SACDIN software component, "Crypto". The relative error, indicative of the degree of fit of the estimating formula to the Crypto data, is tabulated below, and compared with the corresponding figures representing the degree of fit to the Dialog Manager.

Relative Error	Dialog Manager	Crypto
Overall	-.0096	-.0474
Average by Procedure	.027	-.1056
Standard Deviation by	.564	.8917

The relatively good fit of the size estimating formula derived from the Dialog Manager program and applied to the Crypto program supports our contention that the formula is a general one, applicable provided that proper design decomposition rules are followed.

The data suggests that there are relationships between the counts of variables in state machine representations of software designs and the amount of code produced from the design. These relationships can be used to estimate code size based on designs implemented using the state machine technology. The data also suggests a connection between the state machine and Halstead software models.

The formula for the number of SLOC, given above, can be converted to one representing the number of assembly language SLOC, without comments. The expansion ratio of the language in which the SACDIN programs were written is about 1.2, and these programs had about 40% comments. Therefore, S, assembly, without comments is:

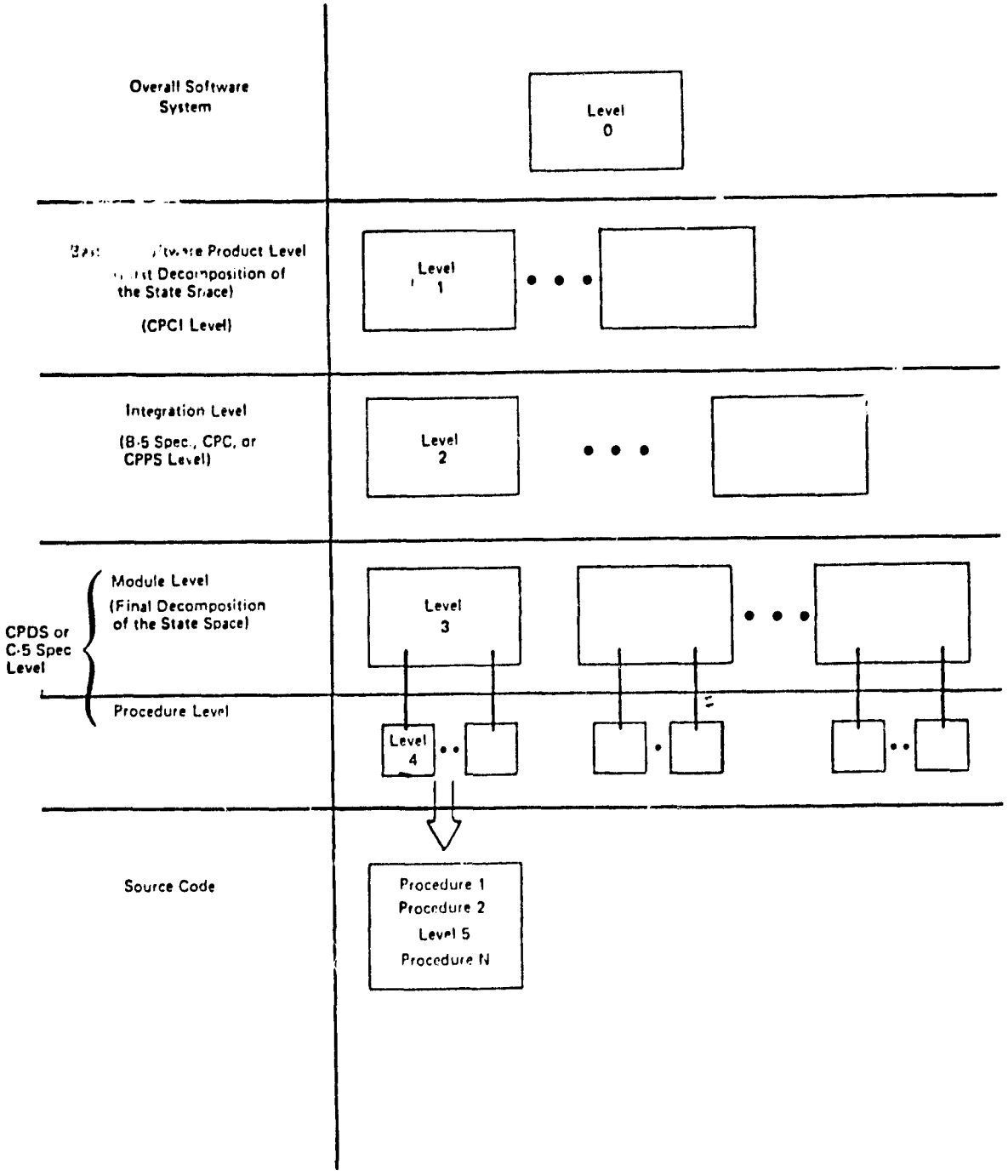
$$S = 8.825 \times 1.2 \times .6 \times V \log_e V = 6.354\ V \log_e V$$

Any software system should be decomposable into 6 "levels", ranging from level 0, the initial program specification, through level 5, the code. The levels are depicted in Figure 2. The formulas presented above were derived for application at level 4, the procedure level. From this point of view of levels, the design and code are essentially more detailed statements of the requirements (the later ones addressed to the machine, while the earlier or higher levels are addressed to people).

Since any software system should have the same number of decomposition or specification levels, a system having more code should have proportionally more "boxes" at each level. Hence, one should be able to produce an estimate based on the number of boxes at a certain level, recognizing that, on the average, about the same amount of function (and hence code count, for a language at a certain level, e.g., assembly) should be resident in a "box" at a given level in the specification hierarchy. A similar notion is used by some

FIGURE 2

Levels of Specification



hardware estimators. Based on experience, a hardware estimator might estimate, for example, that a certain amount of function might require "about 1/2 type x box", where he is familiar with a "type X" box which is an element of an existant system.

Based on the SACDIN data, we note that each level 4 procedure machine has an average of 6 variables, and hence has an average of 68 SLOC (assembly). Also, there is an average of 4 level 4 machines per level 3 machine. Hence, there is an average of 273 SLOC per level 3 machine. Finally, there is an average of 20 level 3 machines per level 2 machine, suggesting an average of 5460 SLOC (assembly) per level 2 machine.

Acknowledgement

The authors express their thanks for the support provided by Mr. Don Zarefoss of IBM, FSD, Gaithersburg, Maryland during the course of the developments described here.

REFERENCES

1. Linger, R. C., Mills, H. D., and Witt, B. I., "Structured Programming Theory and Practice," Addison-Wesley, 1979, pg. 32.

2. Ferrantino, A. B., and Mills, H. D., "State Machines and Their Semantics in Software Engineering, "IEEE COMSAC, Chicago, Fall, 1977.

3. Halstead, M. H., "Elements of Software Science", Elsevier, 1977.

4. Britcher, R. N., and Moore, A. R., "Increased Productivity Through the Use of Software Engineering in an Industrial Environment", "IEEE Computer Society Fifth International Computer Software and Applications Conference"; November, 1981, IEEE Catalog No. 81CH1698-0; pg. 131.

5. Cruickshank, R. D., and Lesser, M., "An Approach to Estimating and Controlling Software Development Costs", in "The Economics of Information Processing", Vol. 2; pg. 139; Springer-Verlag, 1982.

6. Herdan, G., "The Theory of Language as Choice and Change", Springer-Verlag; 1966, pg. 86 and other pages.

7. Henry, S., and Kafura, D. H., "Software Structure Metrics Based on Information Flow", IEEE Transactions on Software Engineering Volume SE-7; Number 5, September, 1981, pg. 510.

8. Albrecht, A. J., "Measuring Application Development Productivity", Proceedings IBM Applications Development Symposium, Monterey, California; October 14-17, 1979; GUIDE International and SHARE, Inc., IBM Corporation, pg. 83.

9. Gaffney, J.E., "Software Metrics: A key to Improved Software Development Management"; presented March, 1981, Pittsburgh, at the conference, "Computer Science and Statistics; 13th Symposium on the Interface"; also proceedings published by Springer-Verlag, 1981.

10. Christensen, K., Fitsos, G. P., and Smith, C.P., "A Perspective on Software Science, "IBM Systems Journal; Vol. 20, No. 4, 1981, pg. 372-387.

11. Savage, J. E., "The Complexity of Computing"; Wiley, 1976, No. 11.

THE VIEWGRAPH MATERIALS

for the

R. BRITCHER/J. GAFFNEY PRESENTATION FOLLOW

ESTIMATES OF SOFTWARE SIZE FROM STATE MACHINE DESIGNS

R. N. BRITCHER
IBM, FEDERAL SYSTEMS DIVISION,
GAITHERSBURG, MD.

J. E. GAFFNEY, JR.*
NATIONAL WEATHER SERVICE
SILVER SPRING, MD.

PRESENTATION AT

SEVENTH ANNUAL SOFTWARE ENGINEERING WORKSHOP
NASA, GODDARD SPACE FLIGHT CENTER
DECEMBER 1, 1982

* ON LEAVE FROM IBM, FEDERAL SYSTEMS DIVISION

- SOFTWARE DEVELOPMENT WORK EFFORT ESTIMATION
- THE STATE MACHINE MODEL
- SOFTWARE SCIENCE/LINGUISTICS BACKGROUND
- STATE MACHINE/SOFTWARE LINGUISTICS EQUIVALENCE

MOTIVATION

- ESTIMATION OF AMOUNT OF FUNCTION PROBABLY MORE DIFFICULT THAN ESTIMATION OF WORK RATES.
- MORE HAS BEEN DONE ON ESTIMATING WORK RATES THAN SOFTWARE SIZE.
- NEED TO QUANTIFY REQUIREMENTS IN TERMS OF LIKELY AMOUNT OF CODE IMPLIED BY THEM.
- SUCCESSIVE REFINEMENT FROM REQUIREMENTS TO CODE SHOULD BE MATCHED BY ESTIMATION PROCESS.

C-5

SOFTWARE DEVELOPMENT
WORK EFFORT
ESTIMATION METHODOLOGY

WORK HOURS = WORK RATE * AMOUNT OF SOFTWARE FUNCTION

SOME MEASURES OF SOFTWARE FUNCTION

- SOURCE LINES OF CODE
- OPERANDS
- STATE MACHINE VARIABLES

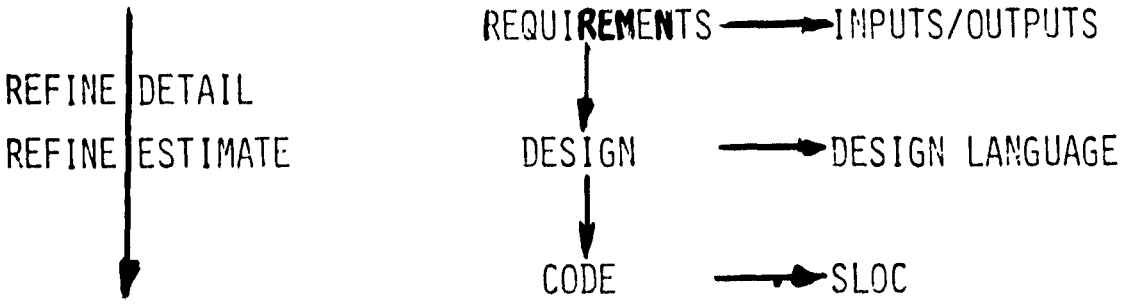
WORK EFFORT ESTIMATION PROCEDURE

- ESTIMATE AMOUNT OF SOFTWARE "FUNCTION"
- ESTIMATE WORK EFFORT

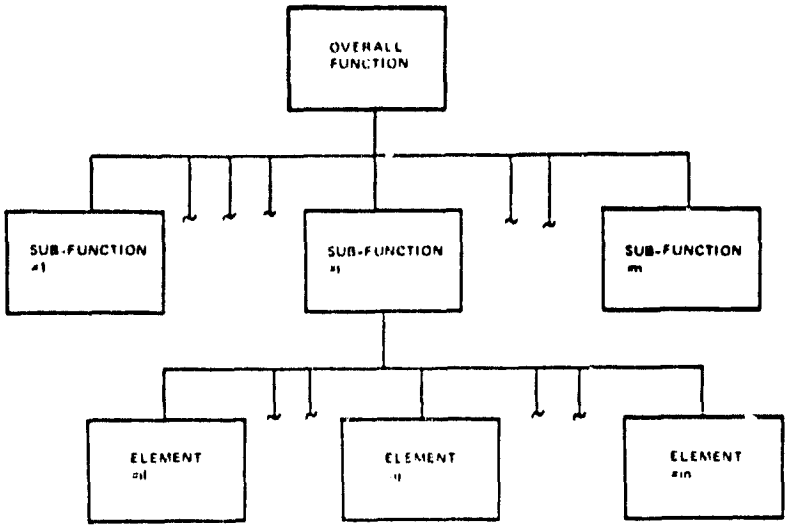
SOFTWARE FUNCTION MEASURES

- LINGUISTIC: REPRESENTS A PROGRAM AS A SEQUENCE OF SYMBOLS, EQUIVALENT TO DISCOURSE
  - SOFTWARE SCIENCE
  - OPERANDS
- STATE MACHINE: REPRESENTS A PROGRAM AS A FUNCTION WITH MEMORY
  - MATHEMATICAL CONCEPT
  - SEQUENTIAL LOGIC
  - VARIABLES

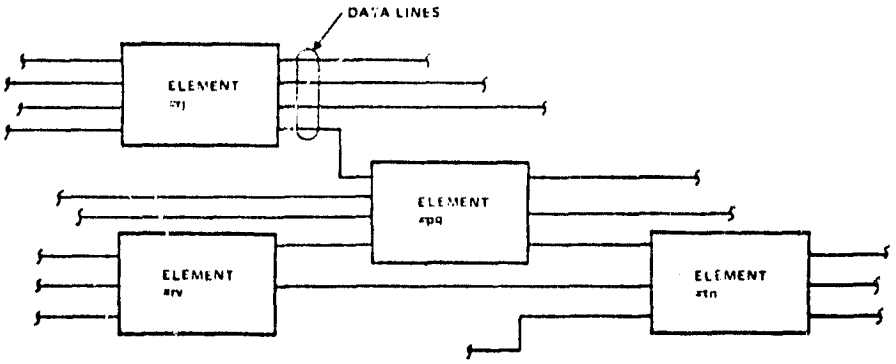
STAGES OF REFINEMENT OF SOFTWARE DEFINITION



FUNCTION DECOMPOSITION

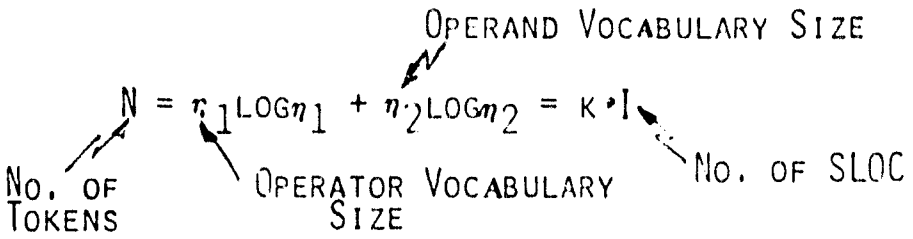


INFORMATION FLOW-NETWORK OF ELEMENTS



HALSTEAD SOFTWARE SCIENCE/LINGUISTICS

MODEL OF A PROGRAM



EXAMPLE:

LA	X
OPERATOR (OP. CODE)	OPERAND (ADDRESS)

$$N = A \cdot \eta_2 LOC \eta_2$$

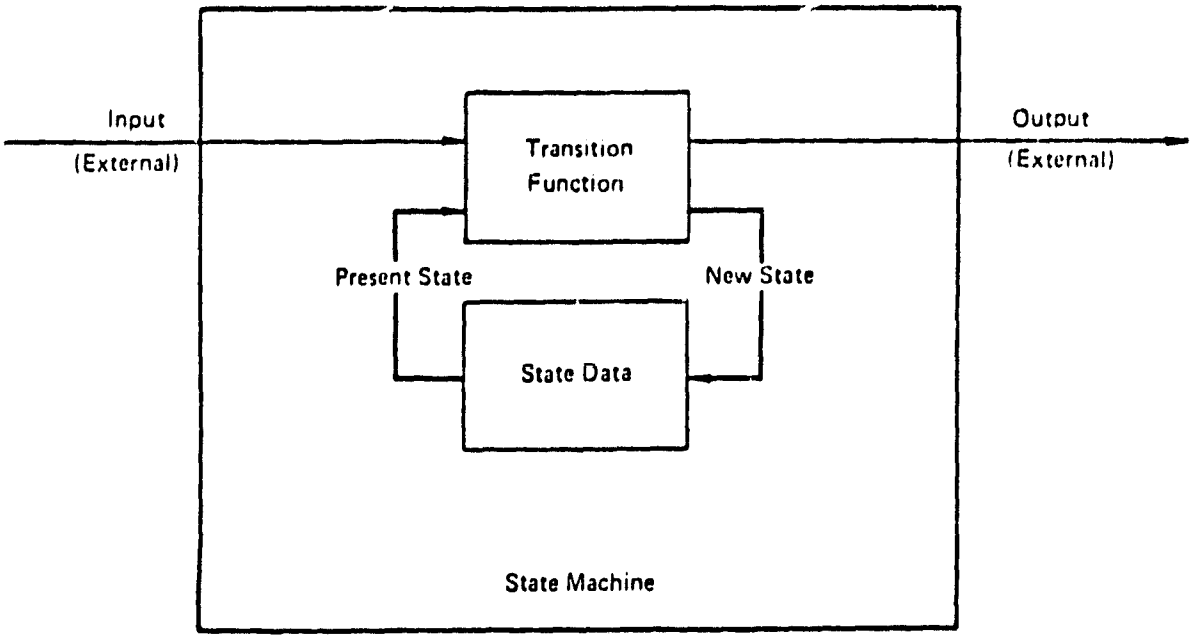
$$\doteq B \cdot \eta^*_2 LOG \eta^*_2$$

η^*_2 = NO. OF INPUTS/OUTPUTS AT ALGORITHM LEVEL

STATE MACHINE MODEL

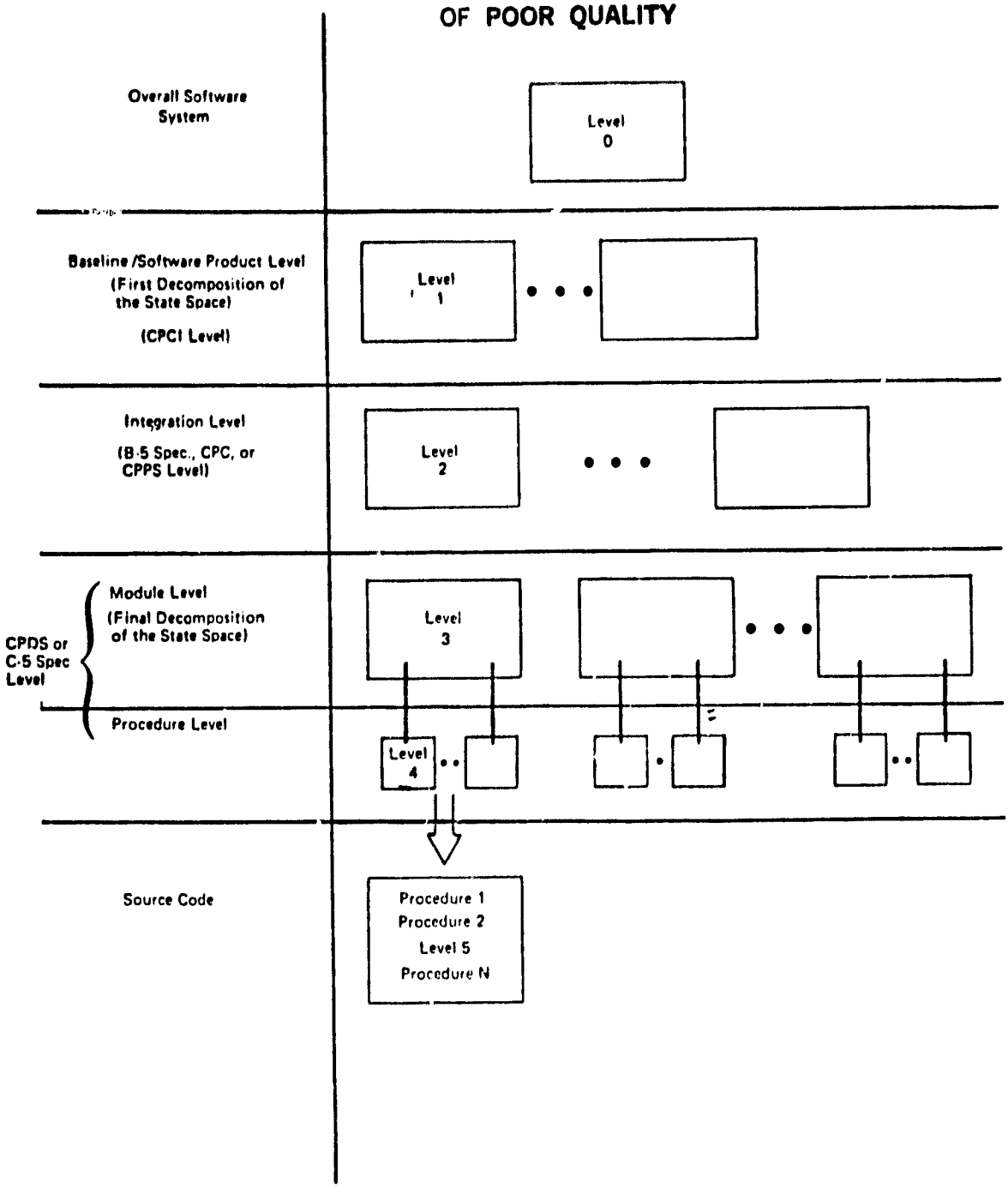
- APPLIES TO PROGRAMS AT VARIOUS LEVELS OF ABSTRACTION
  OVERALL ⟶ INDIVIDUAL PROCEDURE
- APPLICABLE AT SUCCESSIVE LEVELS OF REFINEMENT
- BASED ON THE MEALY-MOORE MODEL OF SEQUENTIAL MACHINES
  DEVELOPED 25 YEARS AGO
- MAPS GENERALIZATION OF "INPUT" (PRESENT PLUS PAST) TO
  "OUTPUT" (PRESENT)

State Machine Representation of a Program



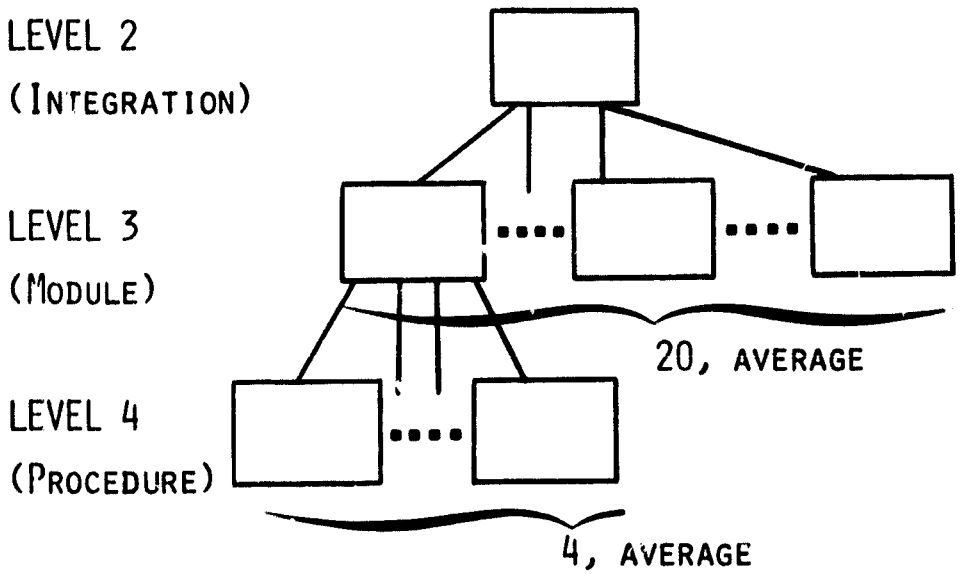
T = [(p. state, input); (n. state, output)]

ORIGINAL PAGE IS
OF POOR QUALITY



Levels of Specification

FAN-OUT OF MACHINES
AT SUCCESSIVE LEVELS OF REFINEMENT OF DETAIL



ESTIMATION METHODOLOGY

- THERE ARE THE SAME NUMBER OF LEVELS, REGARDLESS OF AMOUNT OF CODE
- EARLIER ESTIMATES:
  - DECOMPOSE OVERALL REQUIREMENT INTO SUCCESSIVELY DETAILED STRUCTURE OF "BOXES" AT DIFFERENT "LEVELS"
  - COUNT NUMBER OF BOXES AT LOWEST "LEVEL" OF DETAILING, MULTIPLY BY "AVERAGE" NUMBER OF INSTRUCTIONS.
  - METHOD ANALOGOUS TO HARDWARE "FUNCTION" ESTIMATION BY BOX COUNT, THEN MULTIPLYING BY "AVERAGE" COST OF BOX.
- LATER ESTIMATES:
  - COUNT NUMBER OF VARIABLES PER PROCEDURE
  - APPLY FORMULA FOR EACH PROCEDURE TO GET SIZE ESTIMATE.

STATE MACHINE MODEL ESTIMATING FORMULAS

FOR LEVEL NO.	LEVEL NAME	ESTIMATING FORMULA (ASSEMBLY CODE)
4	PROCEDURE	$6.354\ V\text{LOG}_e V$ (68)
3	MODULE	$25.416\ V\text{LOG}_e V$ (273)
2	INTEGRATION	(5460)

WHERE: V = THE STATE MACHINE "VARIABLE COUNT" (AT THE PROCEDURE LEVEL); IT CORRESPONDS TO HALSTEAD'S η_2, THE "OPERAND" VOCABULARY SIZE.

DEGREE OF FIT OF ESTIMATING FORMULA

RELATIVE ERROR	DEFINING SYSTEM	VERIFICATION SYSTEM
OVERALL	-.0096	-.0474
AVERAGE, BY PROCEDURE	.027	-.1056
STANDARD DEVIATION BY PROCEDURE	.564	.8917

"THE CRUCIAL INGREDIENT OF SCIENCE. THIS IS THE HABIT OF MIND THAT LINKS CURIOSITY WITH DISCIPLINED, RIGOROUS, SUSTAINED INVESTIGATION TO EXPAND THE LIMITS OF KNOWLEDGE".

WILLIAM K. STEVENS
"THE NEW YORK TIMES"
NOV. 9, 1982
PAGE C-1

ATTENDANCE LIST DECEMBER 1, 1982

JON AGDE	ARINC RESEARCH
FREDERDA AKERS	GSFC
DON ANDREW	U S CIVIL SERVICE
PHILLIP ARCANGY	RESEARCH & DATA SYS
ROBERT ARNOLD	UNIV OF MD
EVERETTE AYERS	ARINC RESEARCH CORP
ANGEL BAILEY	CSC
JOE BARKSDALE	GSFC
ALEXANDER BARNES	DM&S
RANDY BARLOW	CSC
VIC BASILI	UNIV OF MD
JIMMY BERRY	USA
JOSEPH BITSUDA	NASA/HQ
MICHELLE BISSONETTE	CSC
CHERYL BITTNER	GENERAL ELECTRIC
DEBORAH BOEHM-DAVIS	GENERAL ELECTRIC
JACK BOND	NSA
DAVID BOON	CSC
PAUL BOOTHROYD	FLEET MATERIAL SUPPORT OFFICE
ROBERT BOROCHOFF	FEDERAL JUDICAL CENTER
RICHARD BREDESON	ITTRI
DALE BRENNEMAN	HUD
A. BRIGGS	CSC
FRED BROOST	CTA
CYNTHIA BROWN	GSFC
VI BROWN	GSFC
DONNA BUCKLAND	REIFER CONSULTANTS
BRIAN BURGER	TEXAS INSTRUMENTS
TOM BURNS	MITRE CORP
JOSEPH BURTON	USDA
JIM CANNING	VPI
DAVE CARD	CSC
JOHN CARL	GSFC
FLOYD CARPENTER	GSFC
JOHN CARSON	GWU
JEFF CHEN	GSC
STEVE CHEUVRONT	CSC
LOUIS CHMURA	NRL
VIC CHURCH	CSC
PAUL CLEMENTS	NRL
GARY COATS	NSA
TED COCHRANE	ITTRI
R. CORTEZ	NASA/HQ

ROBERT CROTCKSHANK	IBM
RAY CURLEY	NSA
BILL DECKER	CSC
DUNCAN DEGRAFFENREID	USDA
CHARLES DICKSON	USDA
JOHN DICKMANS	USDA
DAVID DISATY	CENSUS BUREAU
JIM DORATYS	IBM
DENNIS D. DOE	BOEING AEROSPACE COMPANY
CARL DOHERTINGER	UNIV OF MD
MICKEY DUNTHO	NSA
TOM DUNN	IRS
DAVE ECKHARDT JR	NASA-LANGLEY
BETSY EDWARDS	CSC
WALTER ELLIS	IBM
EUNICE ENG	GSFC
MARY ANN ESFANDIART	GSFC
SUELLEN ESLINGER	CSC
CORTA ETHERIDGE	CONTEL INFO SYS
WILLIAM FARR	NSWC
MARCIA FINERD	SYS DEV CORP
CATHY FRANK	GSFC
YURY FRENKEL	CSC
DAN FRIEDMAN	UNIV OF MD
JOHN GAFFNEY	NATIONAL WEATHER SERVICE
RICHARD J. GALE	TRI-TAC OFFICE
GARY GARR	BURROUGHS CORP
PATRICK GARY	GSFC
CAROL GIAMMO	DCA/CCIC
KEITH GILL	CSC
AMRIT GOEL	SYRACUSE UNIV
JOSEPH GOGUEN	SRI
NANCY GOODMAN	GSFC
A. J. GRACE	IBM
ART GREEN	CSC
ED GREENBURG	JPL
JOE GREGOR	NSA
DICK HAMILTON	BELL LABORATORIES
JOHN HASHMALL	RESEARCH & DATA SYS
ELLEN HERRING	GSFC
DOUG HILLMER	CENSUS BUREAU
TOM HODGSON	MITRE
BARBARA HOLMES	GSC
ADRION HOOK	JPL

RAY HOUGHTON	NATIONAL BUREAU OF STANDARDS
DAN HOWARTH	USDA
WILLIAM HUMPHREY	DOTY
DAVID HUTCHENS	UNIV OF MD
NORMAN IDELSON	ITTRI
RON JABLECKI	DOD
JENNY JACQUES	GSFC
WILLIAM JAMIESON	GSFC
DAVID JOESTING	BENDIX
CHRIS JONES	ITTRI
ROBERT JUDGE	IBM
DENNIS KAFURA	VPI
OWEN KARDATZKE	GSFC
BETH KATZ	UNIV OF MD
FRANCES KAZLAUSKI	NAVDAC
MIRANNE KINGSTON	USDA
BERNARD B. KLEIN	IBM/FSD
RICH KLINKEL	FORD AEROSPACE
JOHN KNIGHT	UNIV OF VA
RICHARD KNOX	CSC
JOHN KOGUT	RESEARCH & DATA SYS
NANCY KRAMER	GSFC
JEFF KUHN	SASC
CHRISTA LAKE	IBM
ROBERT LARSON	USDA
NANCY LAUBENTHAL	GSFC
KAREN LEADER	ITTRI
ERNIE LEE	GSC
GERTRUDE LEE	DOTY ASSOC
RAYMOND LEPESQUEUR	GSFC
KARL LEVITT	SRI
JAY LIEBOWITZ	GWU
ANTHONY MATONE	GSFC
HENRY MAUEC	ITT
NASEEMA MAPOOF	GSFC
JERRY MARSH	ITTRI
THOMAS MASTERS	NSA
J. E. MATHEWS	BENDIX
TOM MARTIN	NSA
ANN MARIE MCCABE	BURROUGHS CORP
W. L. MCCOY	FAA
FRANK MCGARRY	GSFC
MARY ANN MCGARRY	ITTRI
JOHN MCPHEE	DEPT OF COMMERCE

ED MEDEIROS	CSC
REG MEFSON	CTA
PHIL MERWARTH	GSFC
DAVID MICHAEL	FLEET MATERIAL SUPPORT OFFICE
TSAO MIYAMOTO	UMBC
KAREN MOE	GSFC
S. MOHANTY	OSOFT INC
JOHN MUSA	BELL LABS
MATTHEW NADELMAN	CSC
CHRIS NARJUS	NSA
BERNIE NARROW	GSFC
BOB NELSON	GSFC
ROBERT NITCHMAN	FLEET MATERIAL SUPPORT OFFICE
ROBERT NOONAN	COLLEGE OF WILLIAM & MARY
CHARLES OESTEREICHER	MITRE CORP
PAUL ONDRUS	GSFC
TOM OSTRAND	SPERRY/UNIVAC
THOMAS OUSTERIDGE	U S SECRET SERVICE
JERRY PACE	CSC
GERRY PARCOVER	HUD
RAYMOND PAUL	NATIONAL SURFACE WEAPONS CENTER
LEONTE PENNY	USDA
WALTER PENNY	USDA
KARL PETERS	GSFC
JOHN PIETRAS	MITRE
MICHAEL PLETT	CSC
BILL POSTHUMA	GSFC
JERRY PRENTICE	HUD
DOUGLASS PUTNAM	QUANT S/W MGMT
JIM RAMSEY	UNIV OF MD
CHHAYA RAO	GENERAL ELECTRIC
GEORGE RATTE	USDA
GERALDINE RIZZARDI	AFDSC/SFS
SAM REDWINE	MITRE
SALLY RICHMOND	CSC
DON ROBBINS	NSA
MIMI ROBERTSON	ITTRI
JIM ROBINSON	NSA
WILLIAM ROBINSON	SACHS/FREEMAN ASSOC
JOHN ROCCARO	BURROUGHS CORP
MICHAEL ROHLFER	CSC
JORGE LUIS ROMEU	ITTRI
KYLE RONE	IBM
LESLIE RUSHBROOK	IBM/FSD
RON RUTLEDGE	DOT/TSC

JOHN SAPP	SOFTWARE A & E
PAUL SCHEFFER	MARTIN MARIETTA
ROGER SCHOTTEN	BOEING AEROSPACE
LEE SCHUMACHER	MITRE CORP
RICHARD SELBY	UNIV OF MD
PAUL SERAFIN	EG&G
TERESA SHEETS	GSFC
SYLVIA SHEPPARD	GENERAL ELECTRIC
MARIY SHOOMAN	N.Y. POLYTECHNICAL INSTIT
DAVID SIMKINS	IBM
GENE SMITH	GSFC
JOHN SMITH	NSWC
KATHERN SMITH	NASA-LANGLEY
JERRY SNODGRASS	GENERAL DYNAMICS
GLENN SNYDER	CSC
ELLIOT SOLOWAY	YALE UNIV
JOHN SOS	GSFC
MITCHELL SPIEGEL	CONTEL INFO SYS
JOSEPH STAZZONE	OPM
ROBERT STEPHENS	NASA/HQ
T. STEVENS	USDI
TERRY STRAETER	GENERAL DYNAMICS
STEVE SUDOTH	GSC
PAUL SZULEWSKI	DRAPER LAB
KEJI TASAKI	GSFC
ROBERT TAUSWORTHE	JPL
WAYNE TAYLOR	CSC
JAMES TIPPETT	VSA
KENNETH TOM	ARINC RESEARCH CORP
BETSY TURVO	CSC
PAULETTE VAN NORMAN	OPM
M. VILARDO	RESEARCH & DATA SYS
SUSAN VOIGT	NASA-LANGLEY
BRUCE WADDINGTON	BURROUGHS CORP
JACK WAGERTCH	DOT
SHARON WALTGORA	CSC
CHARLES WALLACE	RAYTHEON SERVICE CO
DOLORES WALLACE	NATION BUREAU OF STANDARDS
BARRY WATSON	ITTRI
RON WEEKS	DOD
DAVID WEISS	U.S. NAVAL RESEARCH LAB
ELAINE WEYUKER	COURANT INSTIT
VIRGINIA WILLIAMS	GSFC
PAUL WILLIS	POLYTECHNICAL INSTITUTE
ALICE WONG	DOT

RAYMOND YEH — UNIV OF MD
CHARLES YOUMAN — CFY ENTERPRISES
FRANCOISE YOUSSEFI — UNIV OF MD

SAUL ZAVELER — AFOSC
M. ZELKOWITZ — UNIV OF MD

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-78-002, FORTRAN Static Source Code Analyzer (SAP) User's Guide, E. M. O'Neill, S. R. Waligora, and C. E. Goorevich, February 1978

SEL-78-102, FORTRAN Static Source Code Analyzer Program (SAP) User's Guide (Revision 1), W. J. Decker and W. A. Taylor, September 1982

SEL-78-003, Evaluation of Draper NAVPAK Software Design, K. Tasaki and F. E. McGarry, June 1978

†This document superseded by revised document.

SEL-78-004, Structured FORTRAN Preprocessor (SFORT) PDP-11/70 User's Guide, D. S. Wilson and B. Chu, September 1978

SEL-78-005, Proceedings From the Third Summer Software Engineering Workshop, September 1978

SEL-78-006, GSFC Software Engineering Research Requirements Analysis Study, P. A. Scheffer and C. E. Velez, November 1978

SEL-78-007, Applicability of the Rayleigh Curve to the SEL Environment, T. E. Mapp, December 1978

SEL-79-001, SIMPL-D Data Base Reference Manual, M. V. Zelkowitz, July 1979

SEL-79-002, The Software Engineering Laboratory: Relationship Equations, K. Freburger and V. R. Basili, May 1979

SEL-79-003, Common Software Module Repository (CSMR) System Description and User's Guide, C. E. Goorevich, A. L. Green, and S. R. Waligora, August 1979

SEL-79-004, Evaluation of the Caine, Farber, and Gordon Program Design Language (PDL) in the Goddard Space Flight Center (GSFC) Code 580 Software Design Environment, C. E. Goorevich, A. L. Green, and W. J. Decker, September 1979

SEL-79-005, Proceedings From the Fourth Summer Software Engineering Workshop, November 1979

SEL-80-001, Functional Requirements/Specifications for Code 580 Configuration Analysis Tool (CAT), F. K. Banks, A. L. Green, and C. E. Goorevich, February 1980

SEL-80-002, Multi-Level Expression Design Language-Requirement Level (MEDL-R) System Evaluation, W. J. Decker and C. E. Goorevich, May 1980

SEL-80-003, Multimission Modular Spacecraft Ground Support Software System (MMS/GSSS) State-of-the-Art Computer Systems/Compatibility Study, T. Welden, M. McClellan, and P. Liebertz, May 1980

SEL-80-004, System Description and User's Guide for Code 580 Configuration Analysis Tool (CAT), F. K. Banks, W. J. Decker, J. G. Garrahan, et al., October 1980

SEL-80-005, 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-81-001, Guide to Data Collection, V. E. Church, D. N. Card, F. E. McGarry, et al., September 1981

SEL-81-101, Guide to Data Collection, V. E. Church, D. N. Card, F. E. McGarry, et al., August 1982

SEL-81-002, Software Engineering Laboratory (SEL) Data Base Organization and User's Guide, D. C. Wyckoff, G. Page, and F. E. McGarry, September 1981

SEL-81-003, Software Engineering Laboratory (SEL) Data Base Maintenance System (DBAM) User's Guide and System Description, D. N. Card, D. C. Wyckoff, and G. Page, September 1981

†SEL-81-004, The Software Engineering Laboratory, D. N. Card, F. E. McGarry, G. Page, et al., September 1981

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

†SEL-81-005, Standard Approach to Software Development, V. E. Church, F. E. McGarry, G. Page, et al., September 1981

SEL-81-105, Recommended Approach to Software Development, S. Eslinger, F. E. McGarry, and G. Page, May 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-81-007, Software Engineering Laboratory (SEL) Compendium of Tools, W. J. Decker, E. J. Smith, A. L. Green, et al., February 1981

SEL-81-107, Software Engineering Laboratory (SEL) Compendium of Tools, W. J. Decker, W. A. Taylor, and E. J. Smith, February 1982

†This document superseded by revised document.

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-81-010, Performance and Evaluation of an Independent Software Verification and Integration Process, G. Page and F. E. McGarry, May 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-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-82-005, Glossary of Software Engineering Laboratory Terms, M. G. Rohleder, December 1982

SEL-82-006, Annotated Bibliography of Software Engineering Laboratory (SEL) Literature, D. N. Card, November 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-Related Literature

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

Banks, F. K., "Configuration Analysis Tool (CAT) Design," Computer Sciences Corporation, Technical Memorandum, March 1980

††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)

††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

††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

Basili, V. R., and B. T. Perricone, Software Errors and Complexity: An Empirical Investigation, University of Maryland, Technical Report TR-1195, August 1982

††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

---

††This article also appears in SEL-82-004, Collected Software Engineering Papers: Volume 1, July 1982.

Basili, V. R., R. W. Selby, and T. Phillips, Metric Analysis and Data Validation Across FORTRAN Projects, University of Maryland, Technical Report, November 1982

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

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

††Basili, V. R., and M. V. Zelkowitz, "Operation of the Software Engineering Laboratory," Proceedings of the Second Software Life Cycle Management Workshop, August 1978

††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

††Basili, V. R., and M. V. Zelkowitz, "The Software Engineering Laboratory: Objectives," Proceedings of the Fifteenth Annual Conference on Computer Personnel Research, August 1977

Card, D. N., "Early Estimation of Resource Expenditures and Program Size," Computer Sciences Corporation, Technical Memorandum, June 1982

Card, D. N., "Comparison of Regression Modeling Techniques for Resource Estimation," Computer Sciences Corporation, Technical Memorandum, November 1982

Card, D. N., and M. G. Rohleder, "Report of Data Expansion Efforts," Computer Sciences Corporation, Technical Memorandum, September 1982

††This article also appears in SEL-82-004, Collected Software Engineering Papers: Volume 1, July 1982.

††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

Freburger, K., "A Model of the Software Life Cycle" (paper prepared for the University of Maryland, December 1978)

Higher Order Software, Inc., TR-9, A Demonstration of AXES for NAVPAK, M. Hamilton and S. Zeldin, September 1977 (also designated SEL-77-005)

Hislop, G., "Some Tests of Halstead Measures" (paper prepared for the University of Maryland, December 1978)

Lange, S. F., "A Child's Garden of Complexity Measures" (paper prepared for the University of Maryland, December 1978)

Miller, A. M., "A Survey of Several Reliability Models" (paper prepared for the University of Maryland, December 1978)

National Aeronautics and Space Administration (NASA), NASA Software Research Technology Workshop (proceedings), March 1980

Page, G., "Software Engineering Course Evaluation," Computer Sciences Corporation, Technical Memorandum, December 1977

Parr, F., and D. Weiss, "Concepts Used in the Change Report Form," NASA, Goddard Space Flight Center, Technical Memorandum, May 1978

Reiter, R. W., "The Nature, Organization, Measurement, and Management of Software Complexity" (paper prepared for the University of Maryland, December 1976)

Scheffer, P. A., and C. E. Velez, "GSFC NAVPAK Design Higher Order Languages Study: Addendum," Martin Marietta Corporation, Technical Memorandum, September 1977

Turner, C., and G. Caron, A Comparison of RADC and NASA/SEL Software Development Data, Data and Analysis Center for Software, Special Publication, May 1981

††This article also appears in SEL-82-004, Collected Software Engineering Papers: Volume 1, July 1982.

Turner, C., G. Caron, and G. Brement, NASA/SEL Data Compendium, Data and Analysis Center for Software, Special Publication, April 1981

Weiss, D. M., "Error and Change Analysis," Naval Research Laboratory, Technical Memorandum, December 1977

Williamson, I. M., "Resource Model Testing and Information," Naval Research Laboratory, Technical Memorandum, July 1979

††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

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

††This article also appears in SEL-82-004, Collected Software Engineering Papers: Volume 1, July 1982.

☆ U.S. GOVERNMENT PRINTING OFFICE: 1983-381-791:267