PROCEEDINGS FROM THE FOURTH SUMMER SOFTWARE ENGINEERING WORKSHOP

(NASA-TM-84704) PROCEEDINGS FROM THE FOURTH SUMMER SOFTWARE ENGINEERING WORKSHOP (NASA) 269 p

N82-74123
THRU
N82-74135
Unclas
00/61 09832

HELD ON

NOVEMBER 19, 1979

AT

GODDARD SPACE FLIGHT CENTER
GREENBELT, MARYLAND

National Aeronautics and
Space Administration

PROCEEDINGS

OF

FOURTH SUMMER SOFTWARE ENGINEERING WORKSHOP

Organized by:

Software Engineering Laboratory
GSFC

November 19, 1979

GODDARD SPACE FLIGHT CENTER
Greenbelt, Maryland

FOURTH SOFTWARE ENGINEERING WORKSHOP

November 19, 1979
NASA/GSFC
Building 6, Room S-19

8:20 a.m.	Introduction – F. McGarry, Goddard Space Flight Center
8:30	Panel #1 – 'The Software Engineering Laboratory'
	Participants: F. McGarry, Goddard Space Flight Center V. Church, Computer Sciences Corporation M. Zelkowitz, University of Maryland V. Basili, University of Maryland
10:00	Coffee Break
10:15	Panel #2 – 'Data Collection'
	Chairperson: V. Church, Computer Sciences Corporation
	Participants: P. Belford, Computer Sciences Corporation M. Perie, FAA L. Duvall, IITRI P. de Feo, NASA/Ames
11:00	Panel #3 – 'Experiments in Methodology Evaluation'
	Chairperson: M. Zelkowitz, University of Maryland
	Participants: P. Hsia, University of Alabama S. Sheppard, General Electric W. Fujii, DDI
12:30 p.m.	Lunch
1:30	Panel #4 – 'Software Resource Models'
	Chairperson: F. McGarry, Goddard Space Flight Center
	Participants: B. Cheadle, Martin Marietta Corporation L. Putnam, Quantitative Software Management D. Weiss, NRL
3:00	Coffee Break
3:15	Panel #5 – 'Models and Metrics of Software Development'
	Chairperson: V. Basili, University of Maryland
	Participants: B. Curtis, General Electric J. Musa, Bell Labs A. Stone, General Electric

PANEL #1

THE SOFTWARE ENGINEERING LABORATORY

F. McGarry, NASA Goddard Space Flight Center
V. Church, Computer Sciences Corporation
M. Zelkowitz, University of Maryland
V. Basili, University of Maryland

OVERVIEW OF THE SOFTWARE ENGINEERING LABORATORY

F. McGarry
GSFC

INTRODUCTION

The Software Engineering Laboratory (SEL) is an organization which is functioning for the purpose of studying and evaluating software development techniques in an environment where scientific application software systems are routinely generated to support efforts at the National Aeronautics and Space Administration (NASA). This laboratory has been a joint effort between NASA/Goddard Space Flight Center (GSFC), Computer Sciences Corporation (CSC), Computer Sciences Technicolor Associates (CSTA), and the University of Maryland.

PURPOSE OF THE SEL

Over the past number of years, software costs seem to have been continually increasing in relation to the costs of computer hardware. Because of the vast amounts of resources that have been directed toward the software development problem, there has been a just concern over the overall process of software development.

There certainly are many reasons for the growing concern about the software process. Not only is a sizable portion of government and corporate budgets spent on it, but systems have been getting more complex and software has been required to perform tasks previously considered unattainable. All of this has been due to the rapidly advancing technology in related fields such as computer hardware.

Therefore, in response to these problems, the science of software engineering evolved as a way of developing software through a well-defined process. Using this approach, the software process can be better understood and an attempt can be made to study and improve the product.

Great advances have been made in adding disciplines to this very young science. Over the past 10 years, the advent of disciplined design, development, methodologies, improved management techniques, software metrics and measures, automated development tools, resource estimation models, and many other approaches that have given birth to the term software engineering.

Although numerous software development methodologies have been developed, each claiming to be more effective than the other, it has not been clearly understood (at least as applied to the NASA/GSFC environment) what effects the various methodologies have on various phases of the software development process. More specifically, it has not been understood whether structured, programming, automated tools, organizational changes, resource estimation models, or any of the other technologies would have any effect (either positive or negative) on the software development process at NASA/GSFC. It has also become very clear that it is not easy to define what is a "better" software product. For these reasons (and for several others), the Software Engineering Laboratory (SEL) at GSFC was created.

The SEL set out to accomplish the following two important and valid tasks:

1. To clearly understand the software development process at NASA/GSFC (i.e., how people are used, how money and time are spent, how other resources such as the computer itself are used, how well time lines and milestones are estimated, etc.).

2. To measure the effects of various modern programming practices (MPP's) on the NASA/GSFC software development process.

In order to accomplish these goals, software which was developed for satellite mission support was studied. The Systems Development Section at NASA/GSFC is responsible for generating all flight dynamics support software for GSFC-supported missions. This software includes attitude determination, attitude control, orbit control, and general mission analysis support systems.

The SEL was then used to closely monitor all software developed to support the charter of the Systems Development Section. This includes software developed both by GSFC employees and contractor personnel (primarily Computer Sciences Corporation (CSC)). The SEL was created in the summer of 1976, and it was anticipated that the monitoring process would first of all be done on tasks using conventional means of software development, with various MPPs applied to similar tasks in an attempt to measure the effects of these practices.

Needless to say, the efforts required to accomplish the goals were far more monumental than any member of the SEL ever imagined. Extra efforts were required on nearly every plan of the experiment. Some of the underestimated areas of work include the following:

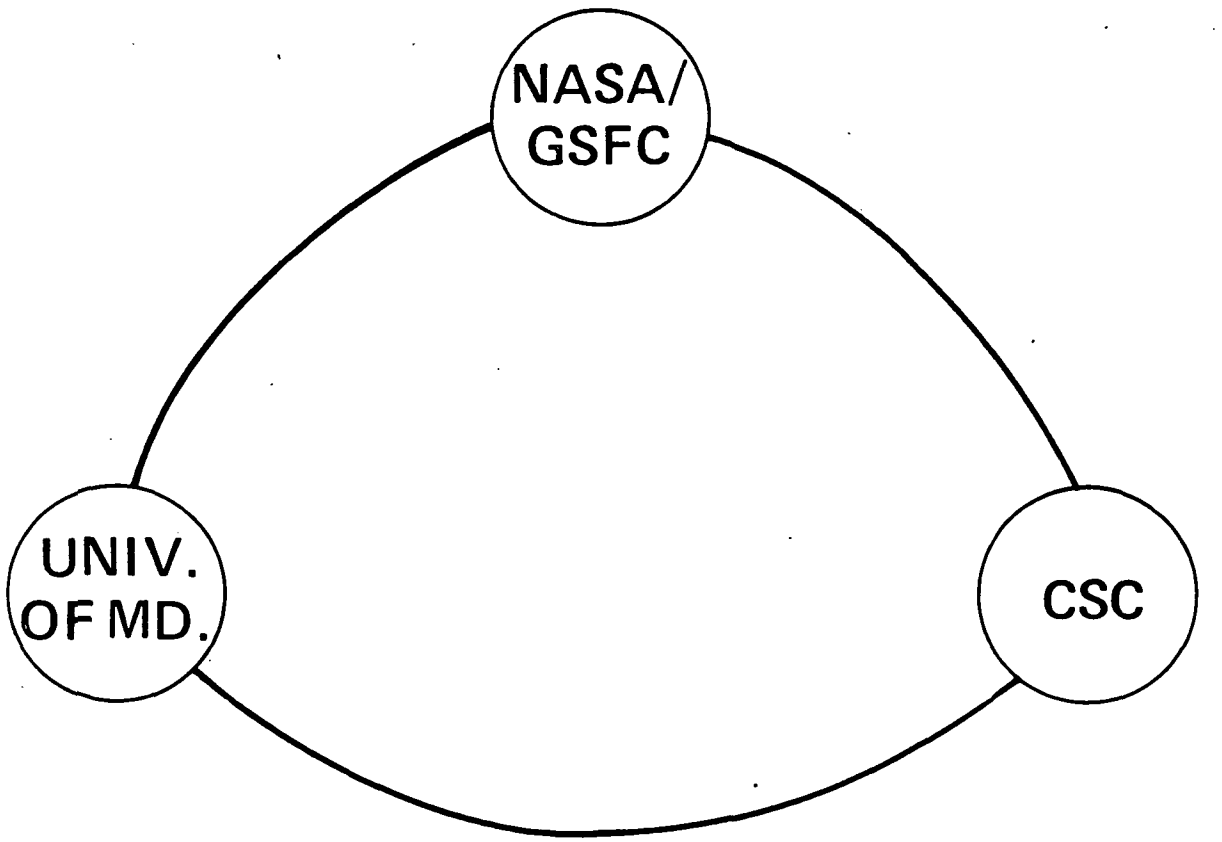
1. Development of clear, understandable data collection forms
2. Organization of the data collection process
3. Design of data storage media for data collected
4. Validation of data made available through data collection forms
5. Design of meaningful, feasible reports that could reflect early results of available data

Through all of the problems and discouraging times that the SEL experienced, the real credit for the success that the lab may have had in the past, and hopefully in the future, must go to the programmers and managers of the tasks involved in the monitoring process. Initially it was felt that a major obstacle was going to be the psychological problem of convincing programmers to accurately provide data on their efforts. However, it was found that not only did quality software people not resent providing the data but they actually made extra efforts to ensure that the data were valid and useful.

The data that have been collected by the SEL cover software development projects starting in late 1976 through 1979. It is anticipated that data will continue to be collected and studied in the future. There have been approximately 25 projects involved, ranging in size from 1500 lines of source to over 120,000 lines of source. Most of the projects were in the 40,000 to 70,000 lines of source category. All of the projects studied were development tasks used to support the flight dynamics area for the Mission Support Computing and Analysis Division (Code 580) at GSFC. The data made available to study the MPPs were collected from a series of forms which were used by all projects. In addition, data were collected through interviews, on-line accounting systems, and by personal inspections of the information by the members of the SEL.

Having investigated projects totaling somewhere around 1 million lines of code, members of the SEL feel that they have been successful in not only gaining insight into the software development process, but also in determining the relative effects of various techniques applied to the software projects.

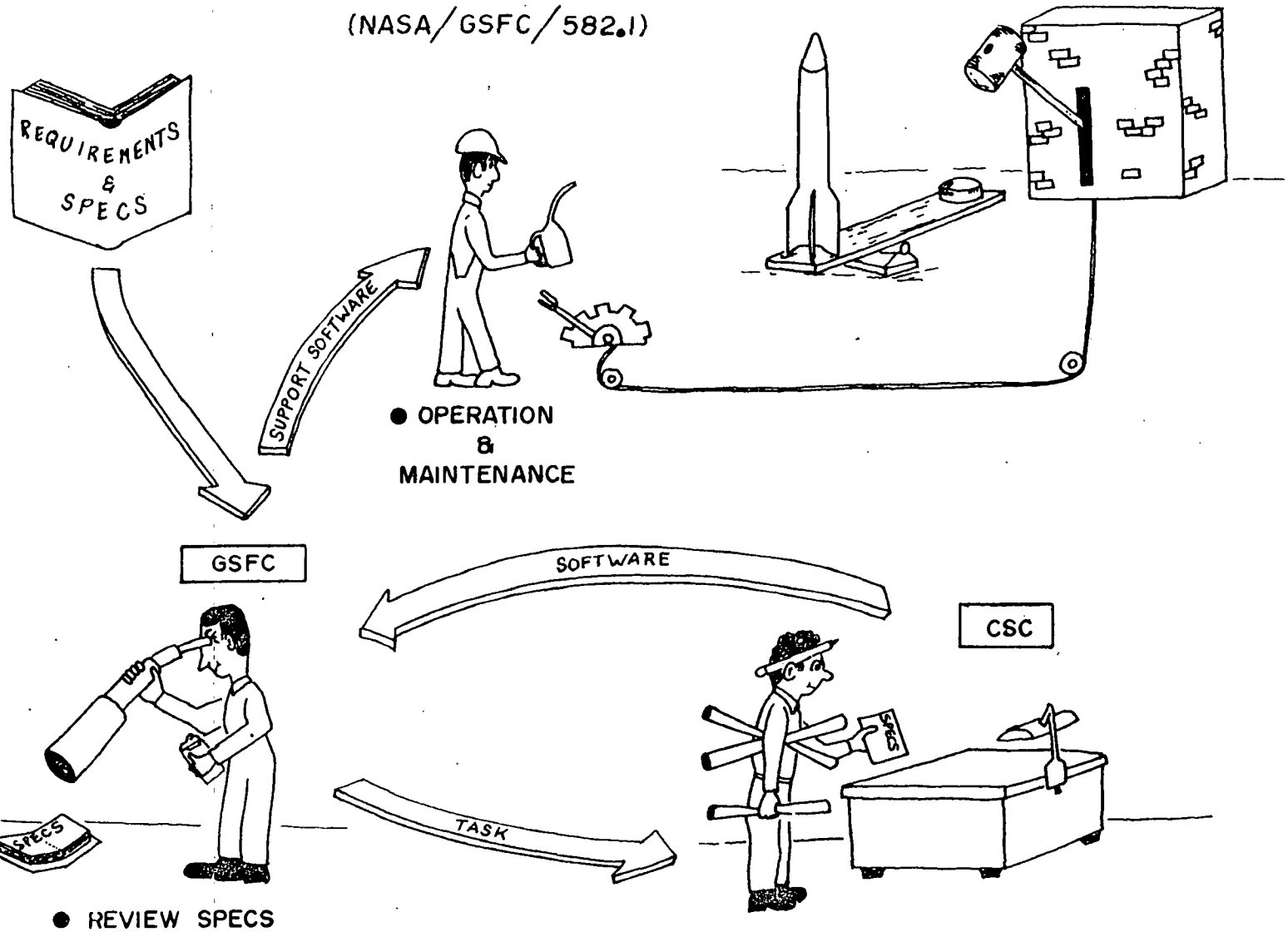
SOFTWARE ENGINEERING LABORATORY



F. McGarry
GSFC

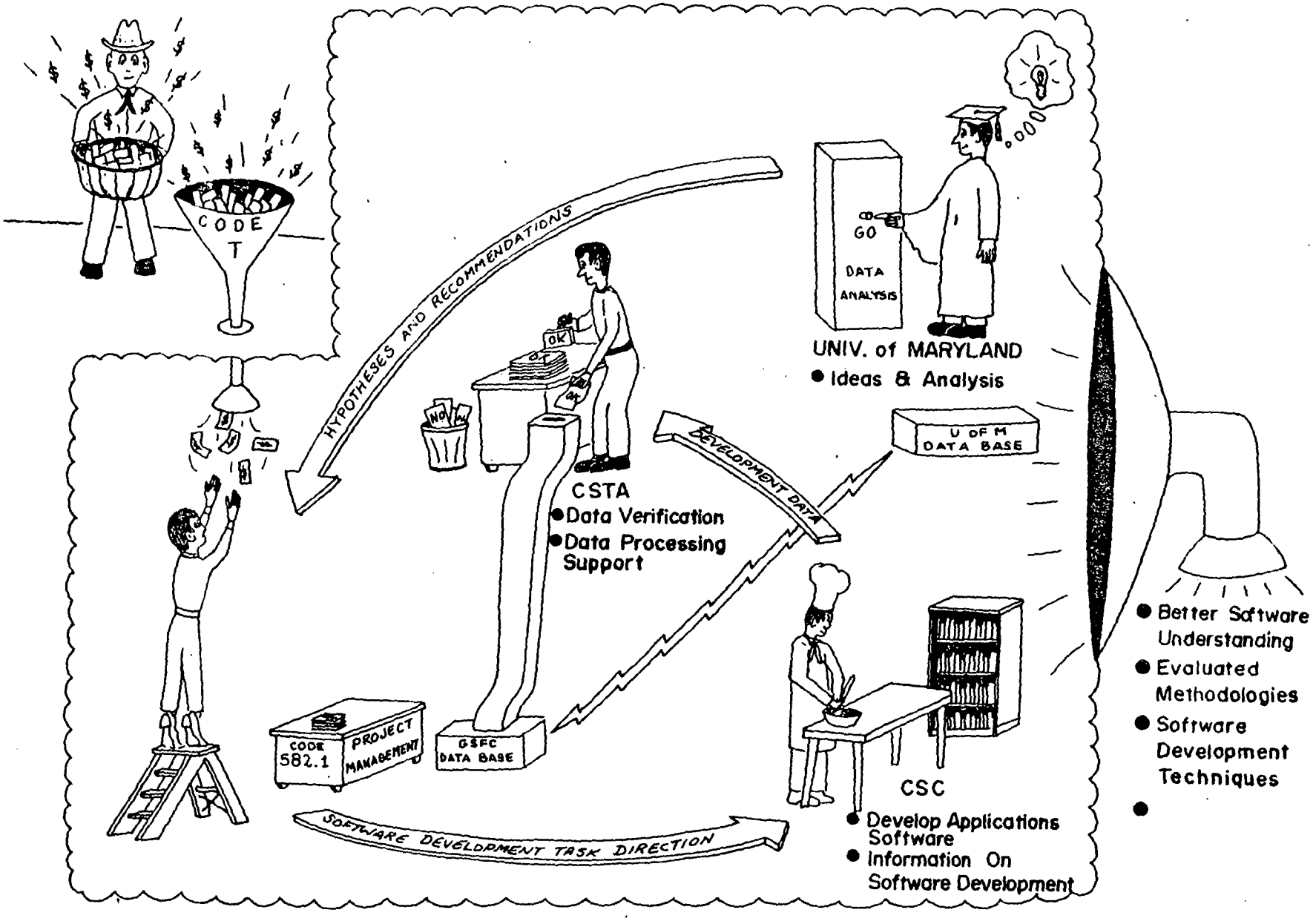
APPLICATIONS SOFTWARE DEVELOPMENT CYCLE

(NASA/GSFC/582.1)



F. McGarry
GSFC
4 of 14

STRUCTURE OF THE SEL



F. McGarry
GSFC
5 of 14

SOFTWARE ENGINEERING LABORATORY – OBJECTIVES

1. UNDERSTAND
   - OUR CURRENT SOFTWARE DEVELOPMENT PROCESS
   - STRENGTHS AND WEAKNESSES
   - TYPE OF ERRORS
   - HOW DO WE SPEND TIME AND MONEY
2. EVALUATE
   - METHODOLOGIES
   - TOOLS
   - MODELS

   } "REAL WORLD" ENVIRONMENT
3. PRODUCE MODEL
   - FOR SOFTWARE DEVELOPMENT
4. IDENTIFY AND APPLY
   - IMPROVED TECHNIQUES

SOFTWARE ENGINEERING LABORATORY – THE PROCESS

- EXPERIMENTS
  - SCREENING (NO PERTURBATIONS)
  - SEMI-CONTROLLED (SPECIFIC METHODOLOGIES APPLIED)
  - CONTROLLED* (TASKS DUPLICATED)
- IDENTIFY INFORMATION REQUIRED
  - FORMS
  - INTERVIEWS
  - AUTOMATIC ACCOUNTING
  - CODE AUDITORS
  - TOOLS (PAN. VALET, . . .)
- ANALYSIS
  - PROFILE INFORMATION
  - APPLY METRICS-MEASURES
  - SOFTWARE MODELING
  - TOOL EVALUATION

F. McGarry
GSFC

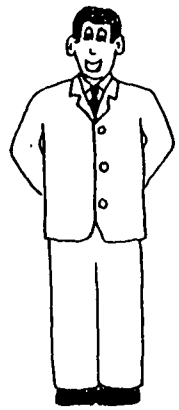
TYPES OF EXPERIMENTS

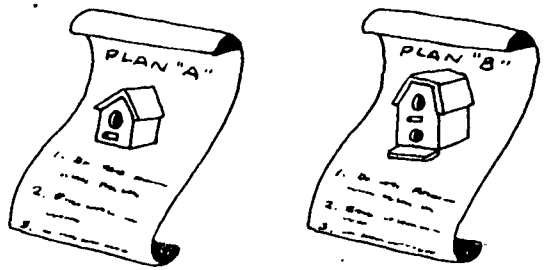
SCREENING TASKS

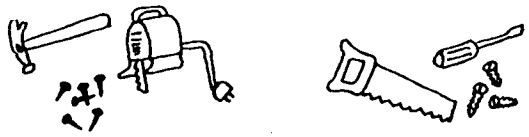
"LET ME WATCH YOU BUILD THE HOUSE."

SEMI-CONTROLLED TASKS

"PLEASE BUILD THE HOUSES AS NOTED IN THE INSTRUCTIONS."

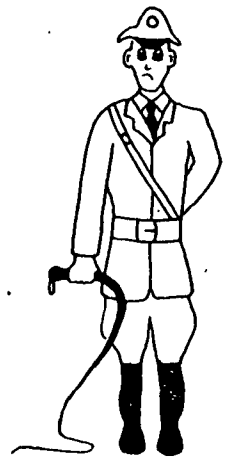


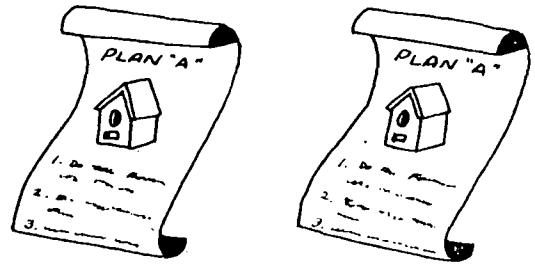


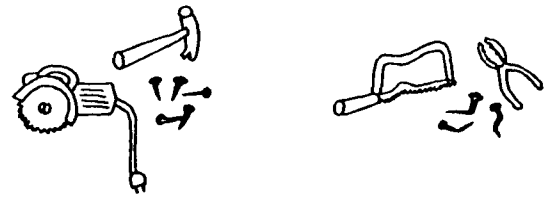


CONTROLLED EXPERIMENT

"YOU WILL BUILD THE HOUSES AS YOU ARE INSTRUCTED."

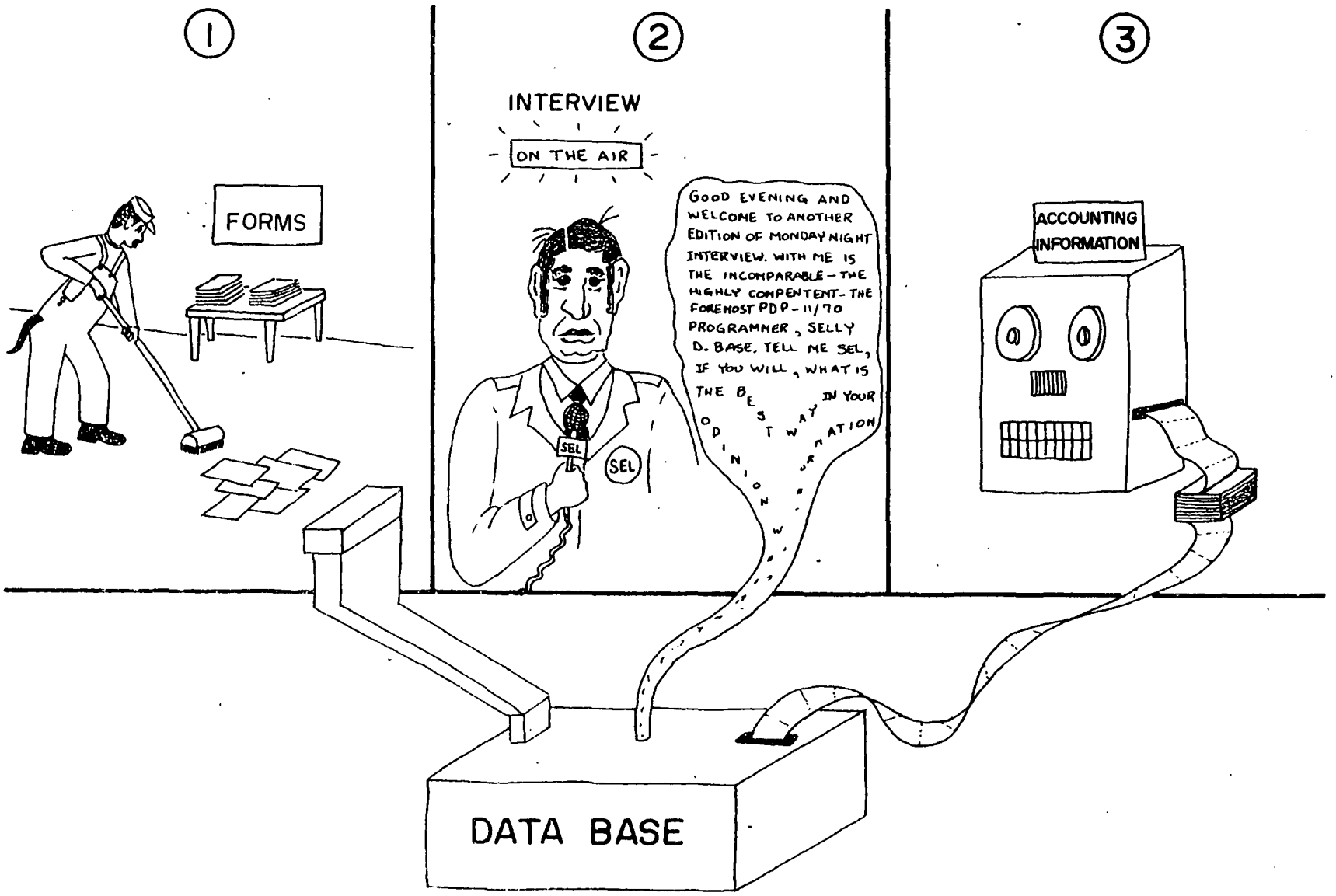






F. McGarry
GSFC
8 of 14

MONITORING THE SOFTWARE DEVELOPMENT PROCESS



SOFTWARE ENGINEERING LABORATORY
DATA COLLECTED

PROJECT	DELIVERED LINES OF SOURCE(K)	TEAM SIZE	DATA COLLECTED METHODOLOGY	COMPLETENESS OF DATA OBTAINED
1	56	7	3-6-7	***
2	3	2	5-6-7	*
3	7	2	2-4-6	**
4	2	2	6	*
5	3	1	2-3	**
6	3	2	3-4	**
7	20	3	2	*
8	62	8	5-6-7	*
9	54	11	1	***
10	70	5	3-4-5-6-7	***
11	3	1	2	***
12	88	11	1-3-4-5-8	***
13	6	2	6	*
14	23	4	6	*
15	6	2	2	*
16	16	2	2	**
17	114	8	1-2-5-6-7	***
18	7	2	6-8	*
19	58	8	1-3-8	***
20	102	10	3-4-5-6-7-8	***

TYPE SOFTWARE

1. SCIENTIFIC
2. UTILITY
3. DATA PROCESSING
4. REAL TIME
5. GRAPHICS

* Assembler Language (All Others FORTRAN)

METHODOLOGY

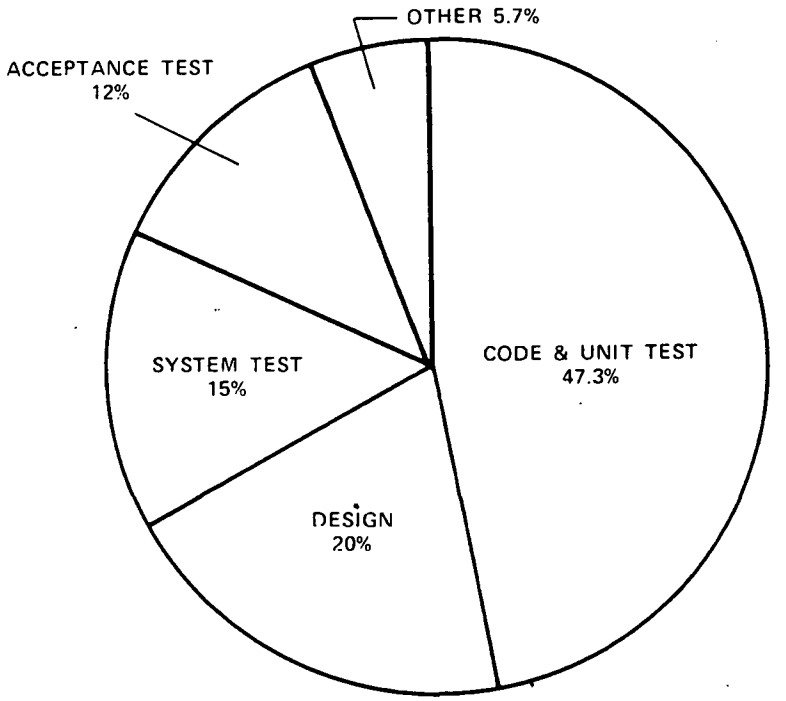
1. CHIEF PROGRAMMER
2. TOP DOWN
3. PRE-COMPILE-STRUCTURE
4. PDL
5. WALK THROUGHS
6. CODE READING
7. LIBRARIAN
8. FORMAL TEST PLAN (DURING DEVELOPMENT)

EXTRACTED DATA

* SOME GOOD DATA
** GOOD DATA
*** VERY GOOD DATA

PROFILE DATA

DISTRIBUTION OF EFFORT BY PHASE



SOURCE: NASA/GSG GSFC (SEL)
AVERAGED 6 PROJECTS (RESOURCE SUMMARY)

PROFILE DATA
EFFORT BY PHASE
(PERCENT)

	TRW	IBM	NASA/GSFC (6 PROJECTS)	NASA/GSFC 1 STUDY TASK COMPONENT STATUS	NASA/GSFC 1 STUDY TASK RESOURCE
CODE	20	30	47	34	50
DESIGN	40	35	20	32	19
CHECKED & TEST	40	25	27	26	19
OTHER		10	6	8	12

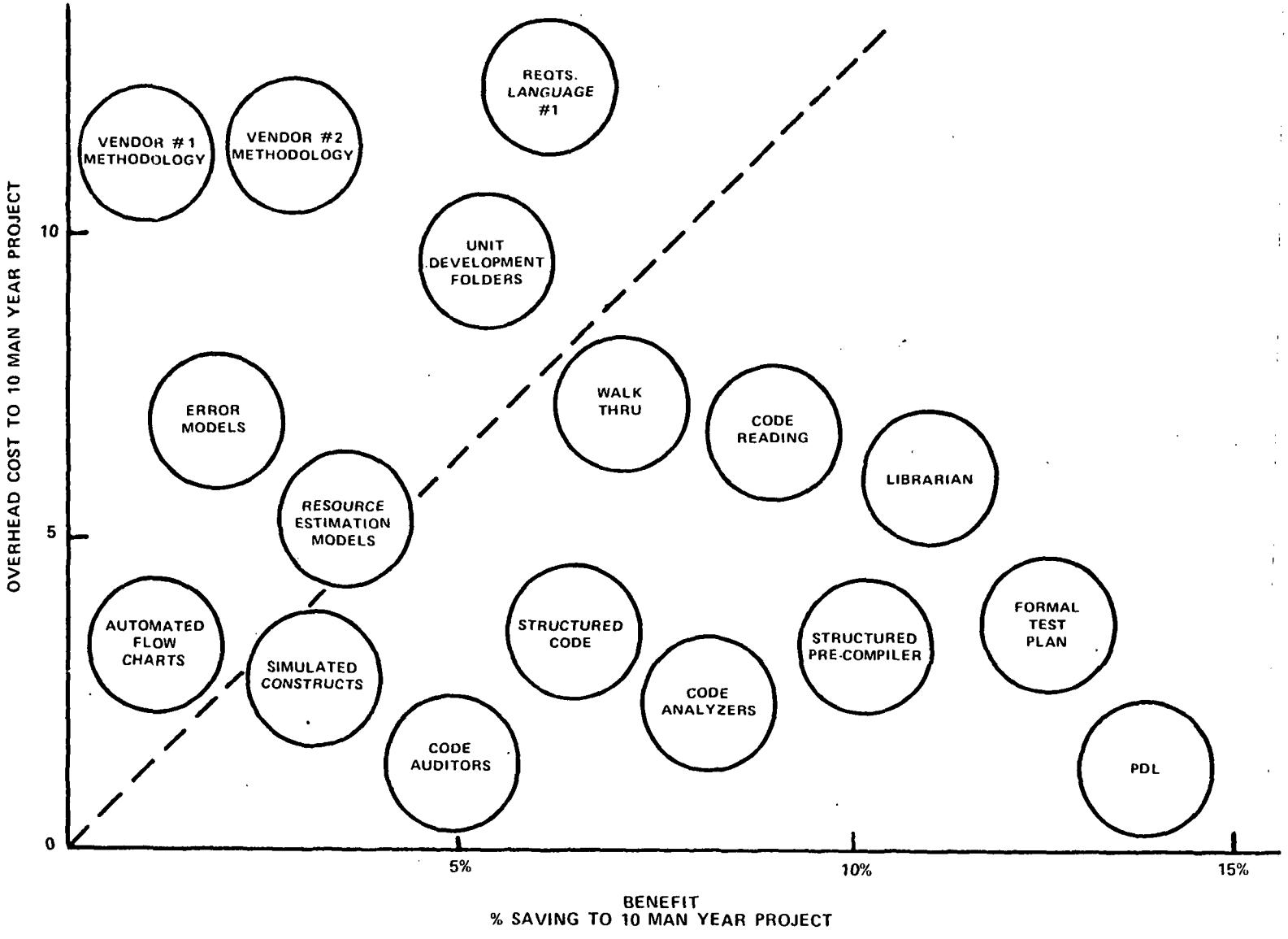
SOFTWARE ENGINEERING LABORATORY

PROJECT NUMBER	(NEW CODE) LINES/ MM	(+20% OLD) LINES/ MM	(NEW CODE) 95 TIME/ 100 LINES	(NEW CODE) 75 TIME/ 100 LINES	(NEW CODE) RUNS/ 100 LINES	% MANAGEMENT	METHODOLOGY FOLLOWED	RESULTS
1	511	512	8.0	24.8	14.9	22.6%	1-3	VERY LATE DELIVERY; EXCEEDED BUDGET; MANY LATE ANOMALIES
2	448	512	6.2	19.6	14.4	14.0%	3	LATE DELIVERY; EXCEEDED BUDGET; MANY LATE ANOMALIES
3	543	546	9.5	20.5	10.5	26.8%	1 8 4 9	ON TIME DELIVERY; WITHIN BUDGET; SOME LATE ANOMALIES
4	715	765	7.4	11.2	14.2	14.5%	3 6 7 9 8 10	ON TIME DELIVERY; WITHIN BUDGET; SMOOTH FINISH
5	504	755	12.7	17.8	13.9	28.7%	1 4 2 5 7 6 8 9	EARLY DELIVERY; WELL WITHIN BUDGET; NO LATE ANOMALIES

TOOLS AND METHODOLOGIES

1 . . . STRUCTURED PRE-COMPILER
2 . . . PDL
3 . . . CHIEF PROGRAMMER
4 . . . UNIT DEVELOPMENT FOLDERS
5 . . . FORMAL TRAINING IN S.E. TECHNIQUES
6 . . . FORMAL TEST PLAN
7 . . . WALK THROUGH
8 . . . CODE READING
9 . . . LIBRARIAN
10 . . . TOP DOWN

SOFTWARE METHODOLOGIES
SUBJECTIVE EXPERIENCE



CONCLUSION

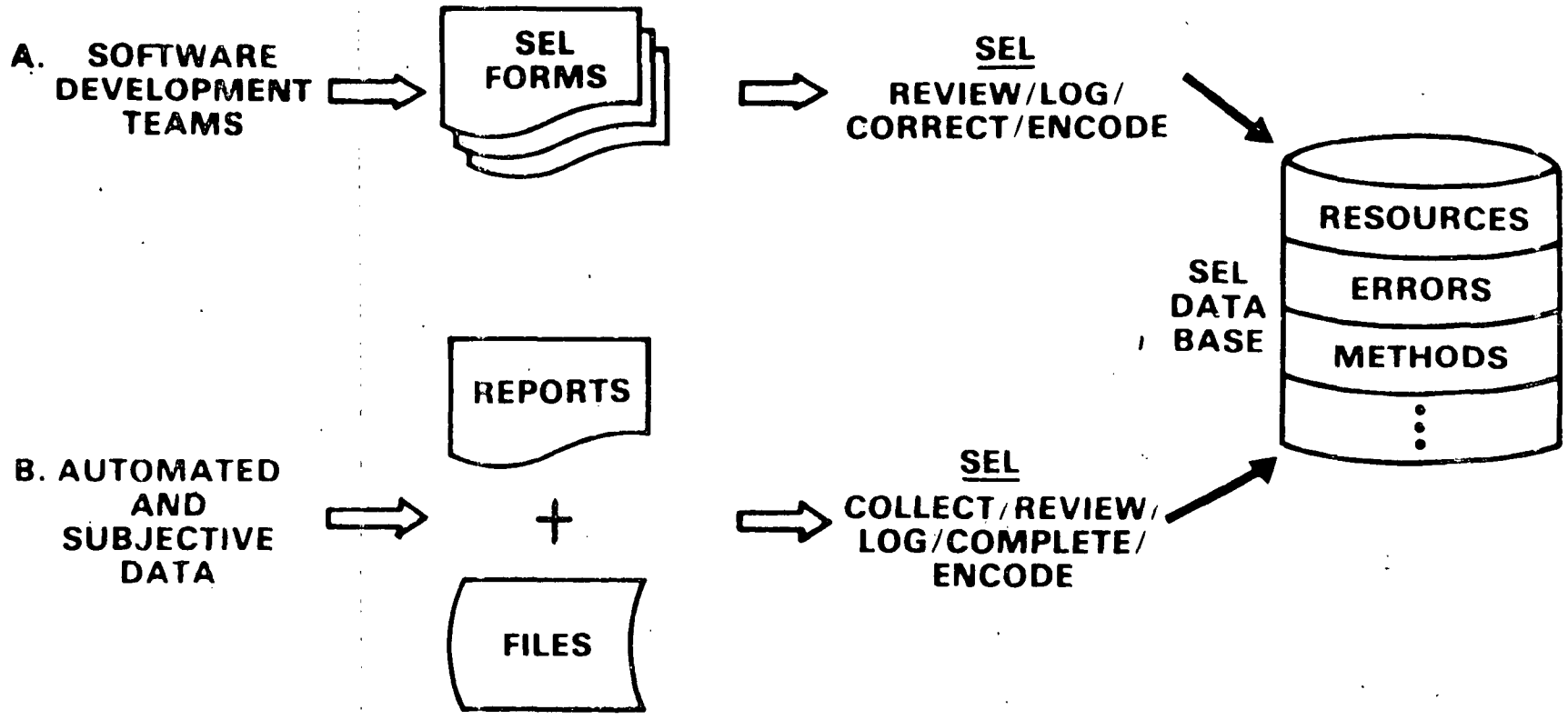
- DATA COLLECTION IS IMPORTANT
- UNDERSTAND THE LOCAL ENVIRONMENT
- COST ABSORBED IN BENEFITS
- THERE ARE MODELS THAT DESCRIBE OUR SOFTWARE ENVIRONMENT
- SOFTWARE TOOL AND METHODOLOGIES DO EFFECT THE SOFTWARE DEVELOPMENT PROCESS
- THE SOFTWARE DEVELOPMENT PROCESS CAN BE IMPROVED
- THERE ARE METRIC THAT DO MEASURE THE "GOODNESS" OF SOFTWARE

SOFTWARE ENGINEERING LABORATORY– THE DATA COLLECTION PROCESS

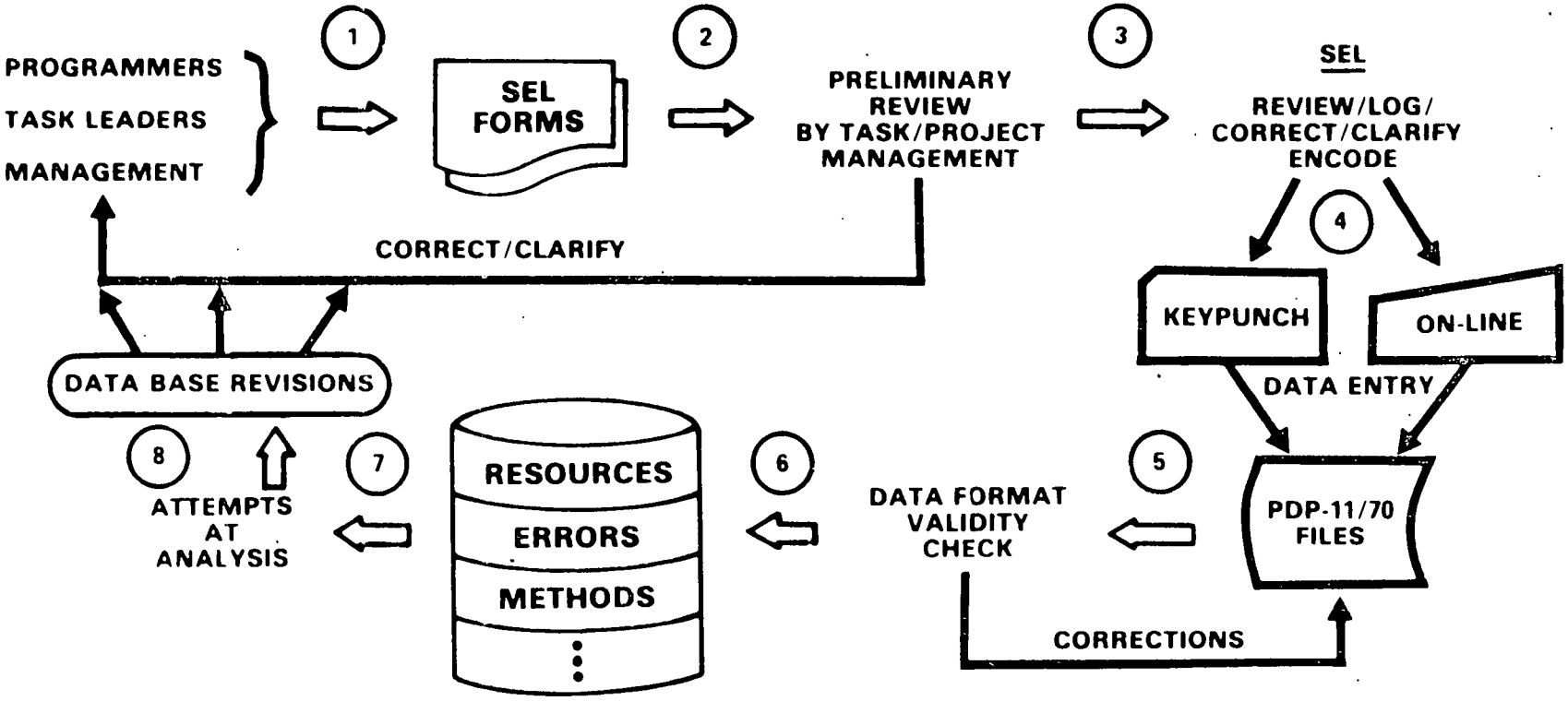
Victor E. Church
CSC

Investigation into the software development activity involves an elaborate process of data collection, review, and preparation for analysis. The overall data flow is described briefly, noting the variety of data sources, the major points of review and data correction, and the macro structure of the resulting data base. Much of the data collection requires the active participation of programmers and software managers involved in the tasks being monitored. Practical considerations of overhead costs, impact on monitored tasks, and programming team reactions and expectations are discussed.

SEL DATA COLLECTION

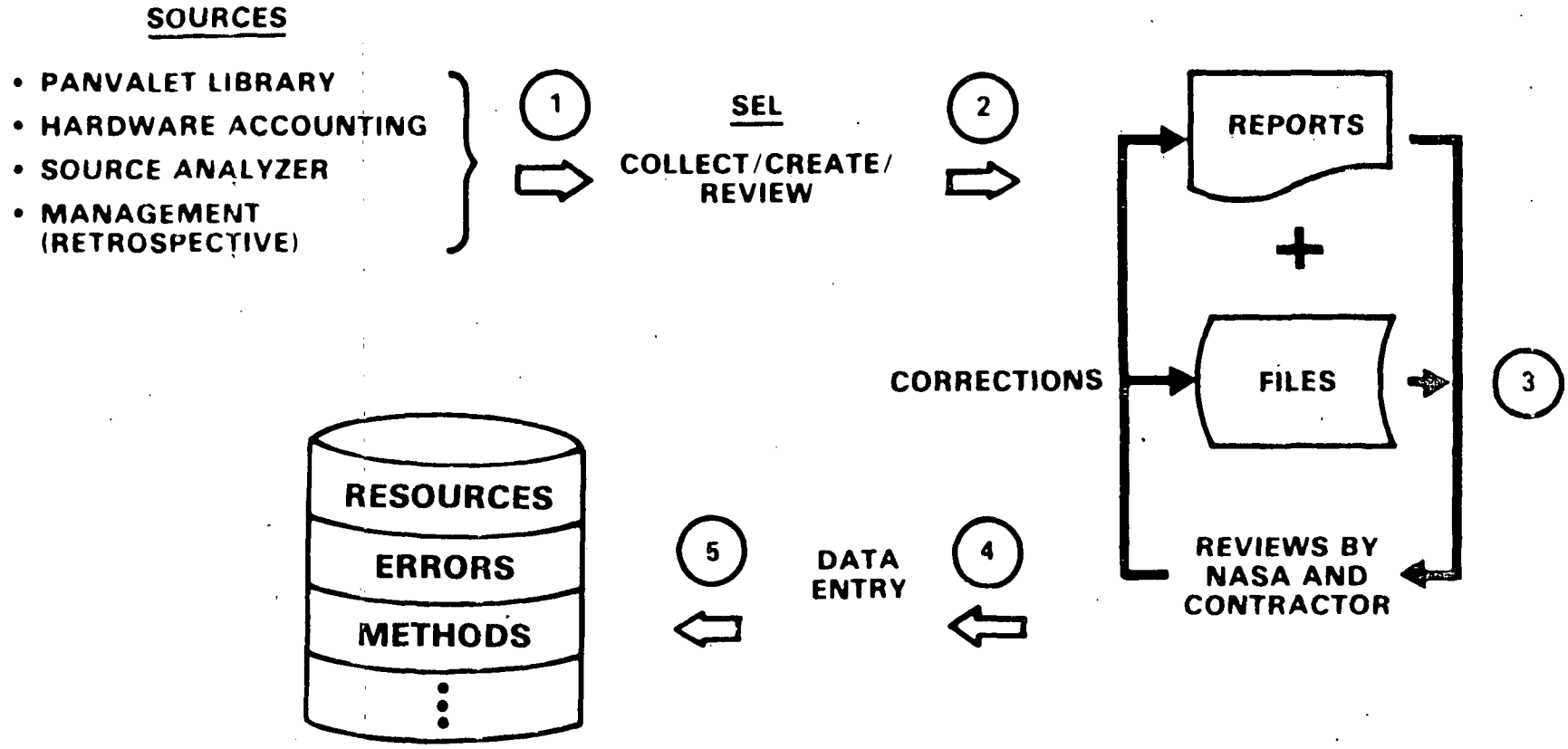


A — TEAM-SUPPLIED DATA

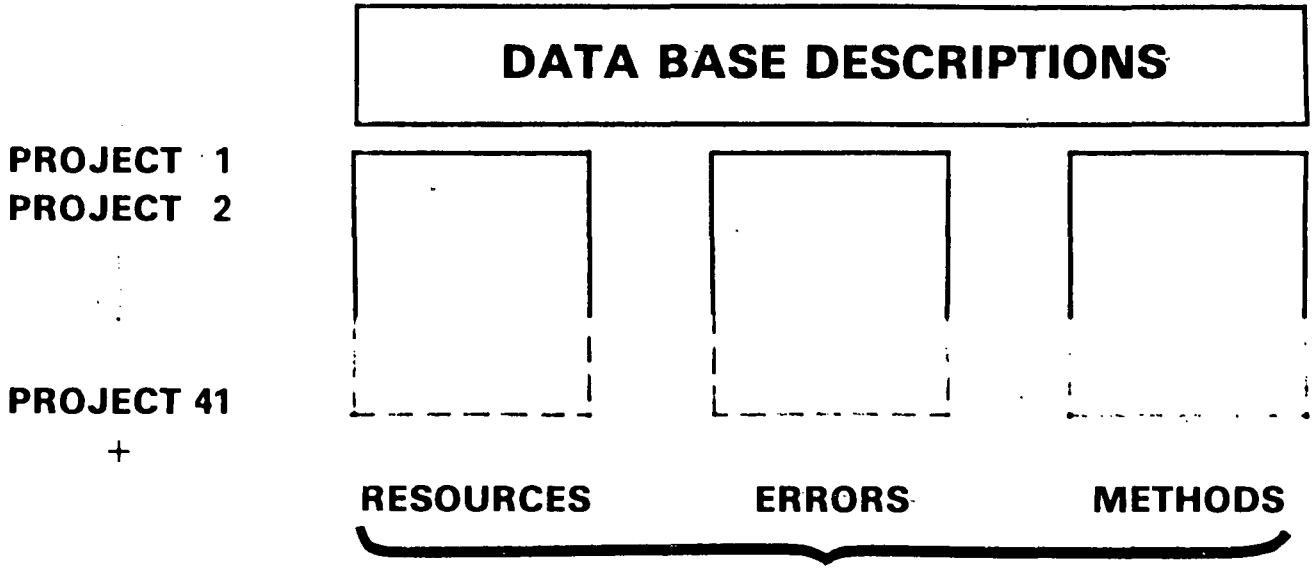


CHU 11-79

B — AUTOMATED AND SUBJECTIVE DATA



DATA BASE ORGANIZATION



ACCESSED BY PROJECT/BY CATEGORY

VIA

- **CUSTOMIZED REPORTING PROGRAMS**
- **DATATRIEVE INTERACTIVE INQUIRY SYSTEM**
- **FORTRAN-CALLABLE USER INTERFACE**

SEL DATA COLLECTION PROCESS — PROBLEMS AND CONSIDERATIONS

- COST TO MONITORED TASKS—10% OVERHEAD
- "HAWTHORNE EFFECT"—NO IMPACT
- PROGRAMMER REACTIONS/CONCERNS:
  - UTILIZATION OF DATA IS NOT VISIBLE OR OBVIOUS TO PROGRAMMERS
  - NO FEEDBACK AVAILABLE TO GUIDE SELF-IMPROVEMENT
  - VALUE OF DATA NOT DEMONSTRATED—CONSIDERED "BUSY WORK"
  - IMPRESSION EXISTS THAT DATA COLLECTION IS BASED ON "WHAT CAN WE MEASURE?" RATHER THAN "WHAT DO WE WANT TO KNOW?"
- PROBLEMS WITH COLLECTED DATA
  - DIFFERENT PROJECTS PRODUCE VARYING LEVEL, CONSISTENCY OF DATA
  - Q/A PROCESS (DURING DATA COLLECTION) HAS TOO LOW A PRIORITY
  - TIMELINESS OF DATA COLLECTION COULD BE IMPROVED

CHU-11-79

DIRECTIONS—

- PROVIDE PERIODIC (MONTHLY, QUARTERLY) REPORTS TO SOFTWARE DEVELOPMENT TEAMS ON ANALYSES OF GENERAL INTEREST, AND OF SPECIFIC INTEREST TO INDIVIDUAL TEAMS AND PROGRAMMERS

- PROVIDE IMPROVED Q/A, ESPECIALLY AT THE TIME AND PLACE OF DATA COLLECTION

- DEVELOP AUTOMATED PROCEDURES FOR THE COLLECTION OF AS MUCH DATA AS POSSIBLE; SHIFT THE OVERHEAD FROM PROGRAMMING STAFF TO HARDWARE

- PROVIDE TRAINING IN MPP AND DISCUSSIONS/ IMPLEMENTATIONS OF THE FINDING OF THE SEL. DEMONSTRATE THE RELEVANCE AND APPLICABILITY OF SEL RESEARCH

CHU-11-79

SOFTWARE ENGINEERING LABORATORY: DATA VALIDATION

Marvin V. Zelkowitz and Eric Chen
Department of Computer Science
University of Maryland
College Park, Maryland 20742

The need to validate the data being collected by the Software Engineering Laboratory is a primary prerequisite before analyses can be attempted. In terms of validation, three phases have been identified: (1) forms validation, (2) project validation, and (3) completeness and consistency validation.

Forms validation is a process that verifies that the data on the forms that are being collected is accurately transferred to the computerized data base. It is mostly a clerical process as the forms are typed into the computer. Minimal checking of data across forms is attempted – all checking is at the local level. In addition, once a project's forms has been entered, the data is rechecked against the original forms before being used in analysis.

Project validation tests whether the entire set of forms for a project is consistent. For example, does the number of hours specified on the resource summary (filled out by the project manager weekly for all project personnel) agree with the number of hours specified by each programmer on the component status report (giving the hours spent each week on each component)? What date is missing (e.g., which reports are not in the data base)? This is a relatively straightforward check on the total collected data from a project.

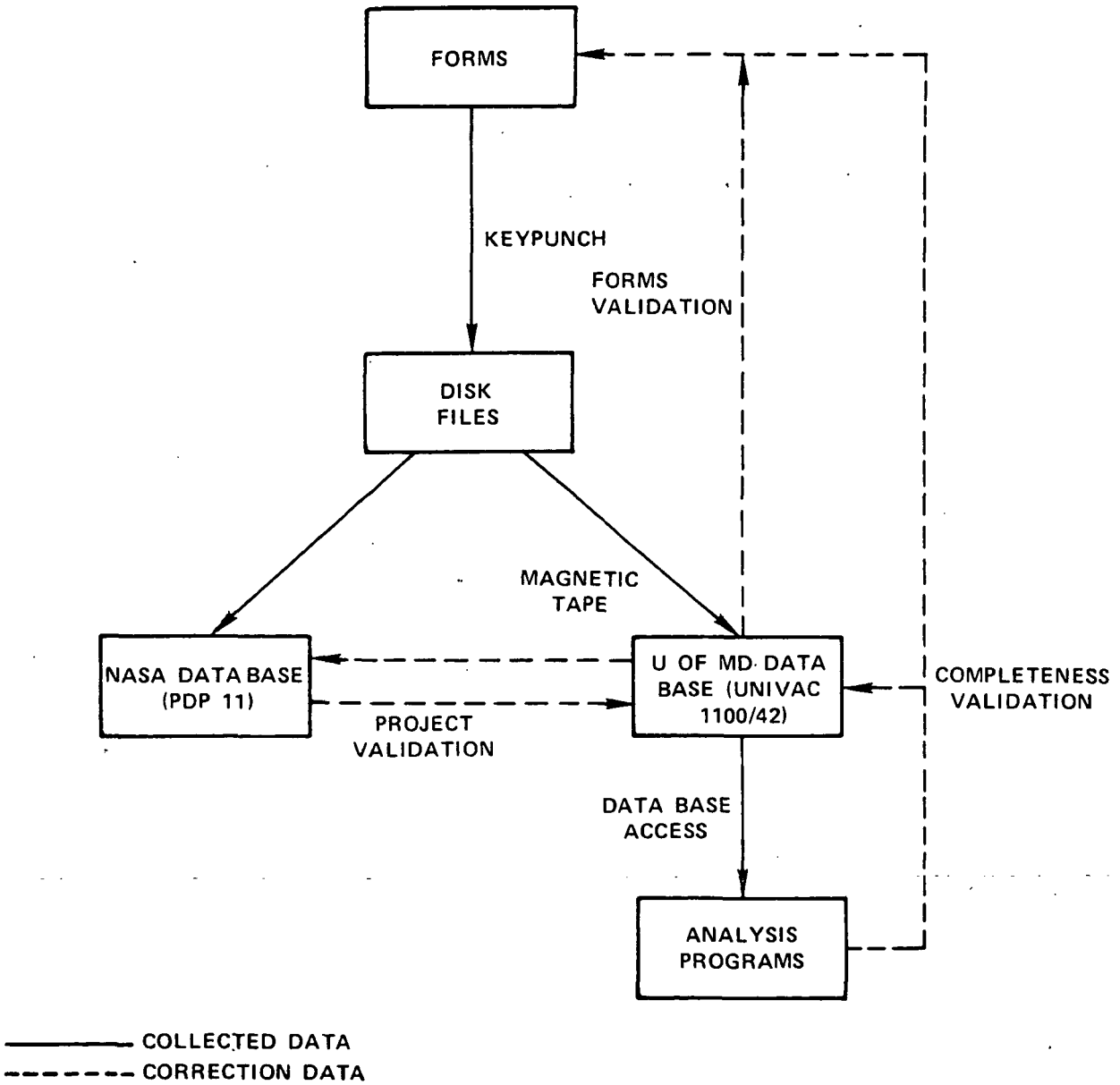
The more interesting question is completeness and consistency validation. This attempts to determine if there is any underlying structure or biasing in the ways forms are being filled out.

The initial approach is to use cluster analysis. Each of the forms is represented as a multidimensional vector of M dimensions. Each vector is projected onto a N–dimensional space using a subset of the M components as a basis. It is determined which forms cluster near one another in this N–dimensional space – such forms being considered related according to the basis chosen. Various regression techniques are being used to see if any of the other (M–N) attributes are predictors of this clustering.

Some of the issues being initially investigated include: Is the programmer identification a predictor of the cluster? (It shouldn't be.) If so, then some of the programmers fill out the forms in certain characteristic ways which would show a biasing in the collected data. On the projects so far checked, this does not seem to be the case. Another question: Is the project name a good predictor when several projects are considered together? If so, then either there is biasing at the project level, or else different methodologies on different projects lead to different data being collected. If true, then the reasons will be investigated. A third initial question to be studied is: Are the clusters indicative of certain characteristic errors? Can clustering be used as an error classification?

While the work is still very preliminary, the use of such clustering techniques in this environment seems promising.

DATA FLOW THROUGH THE SOFTWARE ENGINEERING LABORATORY



M. Zelkowitz
Univ. of Md.

FORMS COLLECTED

Resource Summary (by management)
hours/week/programmer

Component Status Report (by programmers)
hours/week/component/phase

Change Report Form (by programmer)
Each change or error, when found

Computer Run Analysis (by programmer)
Each computer run

General Project Summary (by management)
Each project

Component Summary (by programmers)
Each piece of system

DATA VALIDATION

1. **Forms Validation** – Each form is self-consistent, as it is entered into bases. Checks are both manual and automated.

2. **Project Validation** – Similar data on different forms for a project is analyzed, for missing or incomplete data.

3. **Consistency Validation** – Checks whether there is any systematic biasing of the set of collected forms between projects.

   then –

   Either:

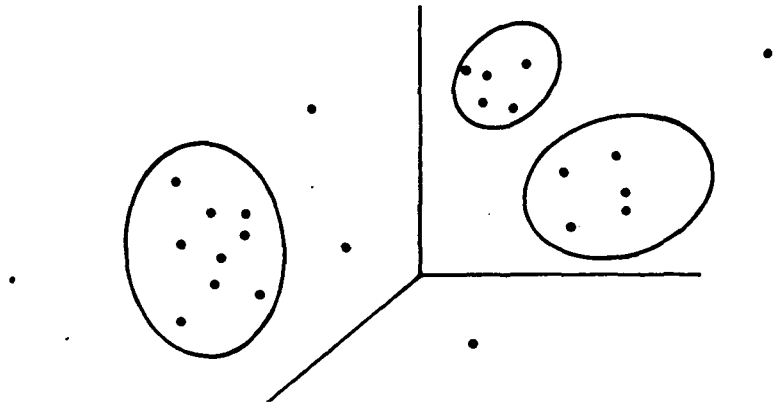
   (a) Projects are not using same interpretation of instructions when filling out forms.

   or (b) Methodology used leads to characteristic differences in approaches to forms.

M. Zelkowitz
Univ. of Md.

CONSISTENCY VALIDATION

- Basic approach uses cluster analysis.
- Each form a multidimensional vector of N dimensions.
- Choose M of those components.



- Objects near one another are "related" by M chosen elements are in same cluster.

Question: Is any one of the (N–M) remaining components a predictor of cluster?

M. Zelkowitz
Univ. of Md.

CLUSTERING ALGORITHM

1. Compute similarity between vectors (forms) I and J. Call it Sij. Sij will have a value between 0 and 1.

2. Choose some threshold B between 0 and 1.

3. If Sij > B then I and J are similar, so set Dij = 1. Otherwise set Dij = 0.

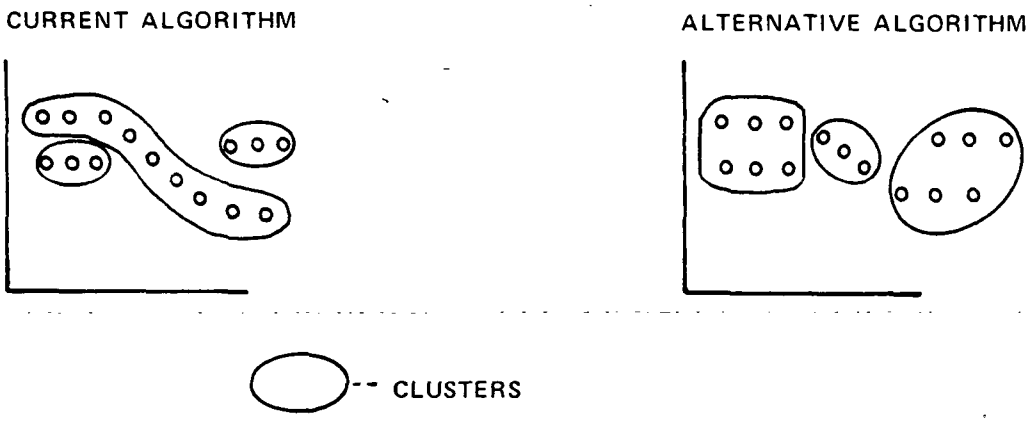
4. When viewed as a graph, Dij = 1 represents that node I is connected by an arc to node J. Compute transitive closure $D^* = D + D^2 + \ldots D^n$.

5. D*ij = 1 if and only if nodes I and J are in the same connected subgraph. These connected subgraphs represent similar forms.

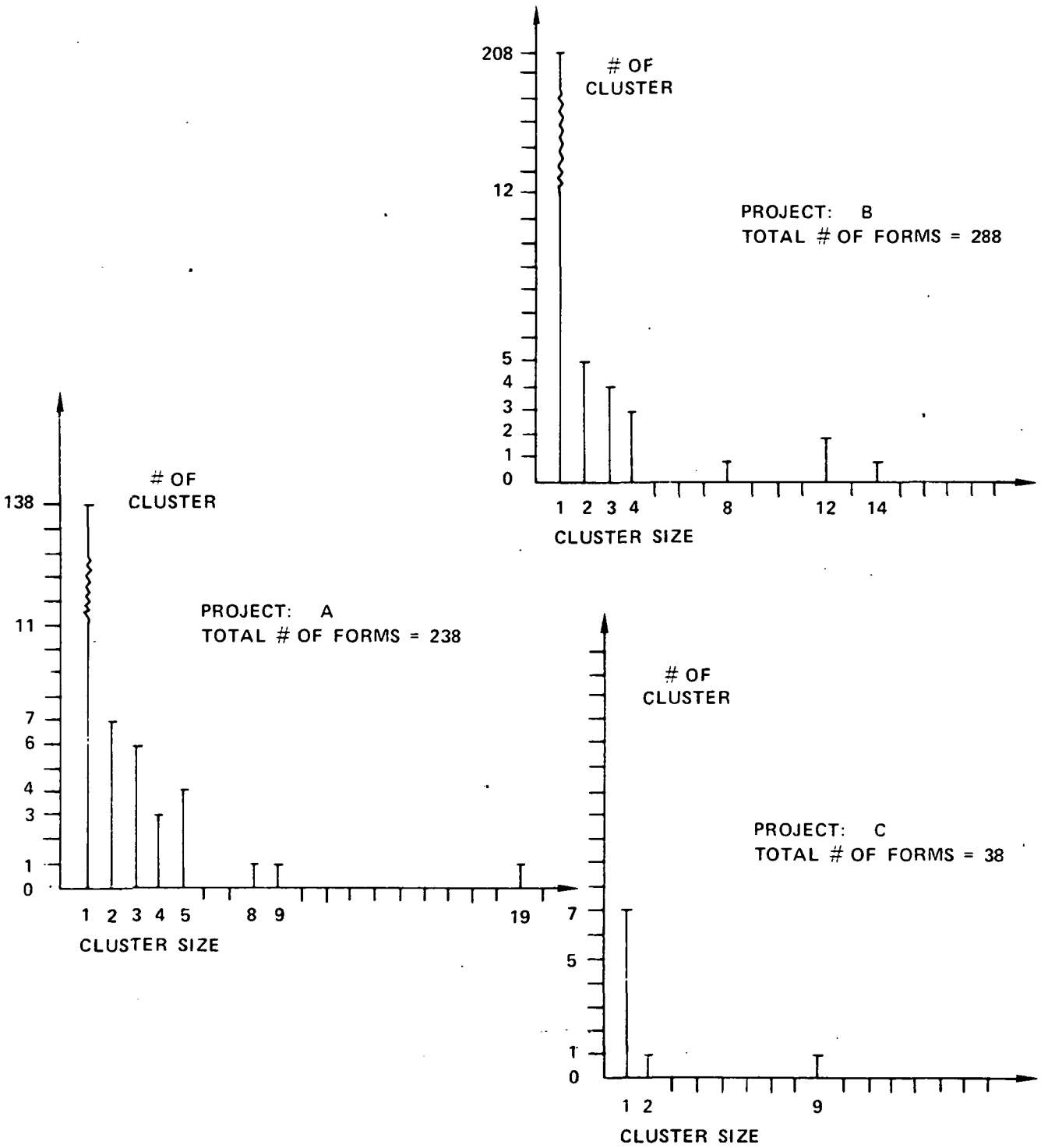
RESEARCH IDEAS

1. Vary B and measure effects on cluster sizes. The larger the B, then the fewer the vectors that will be similar. For the following graphs, B = 0.950.

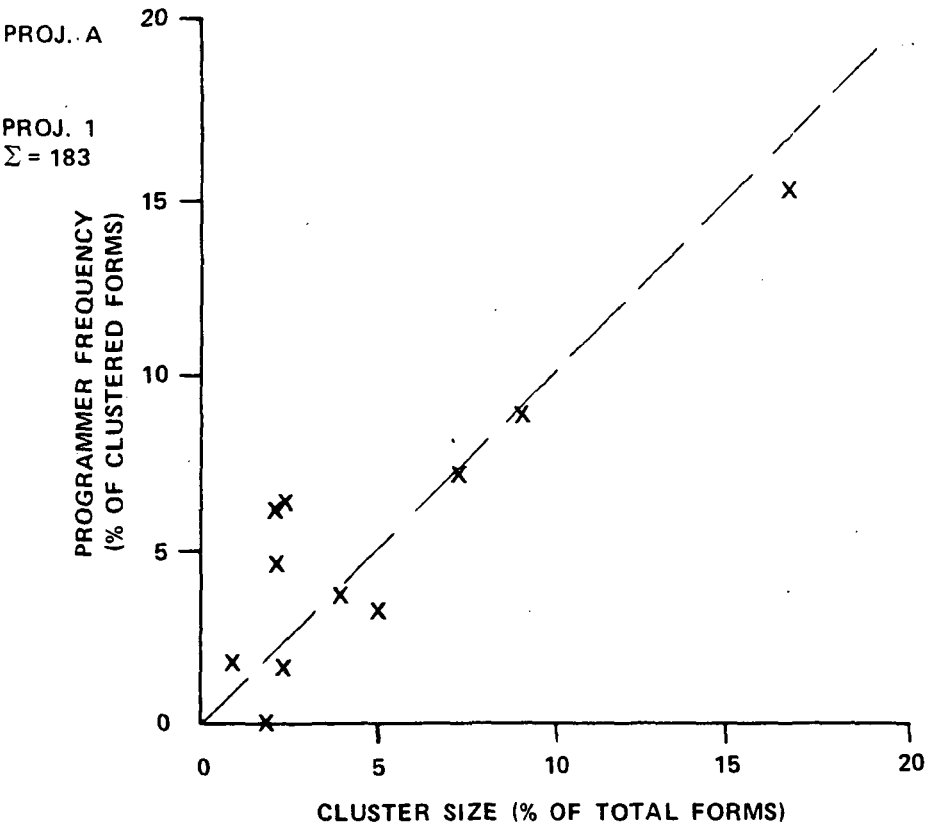
2. Vary clustering algorithm. Current algorithm computes dot product of unit (normalized) vectors. Alternative strategy is to compute clusters as those vectors closest to some centroid instead of simply within the same connected subgraph.

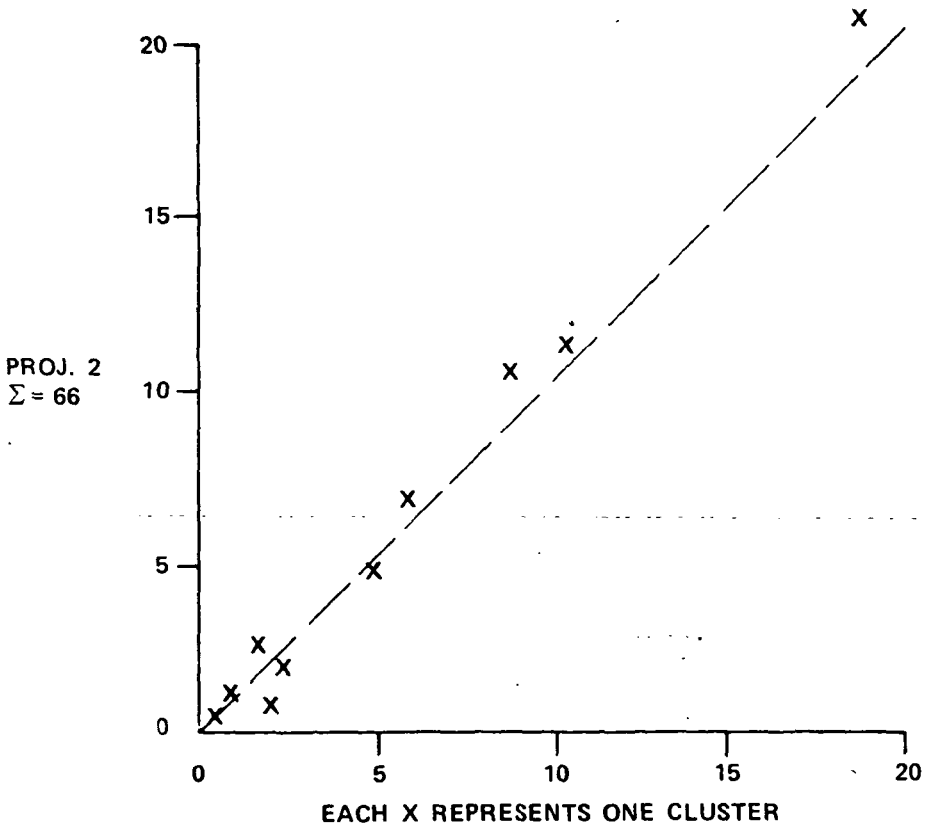


M. Zelkowitz
Univ. of Md.



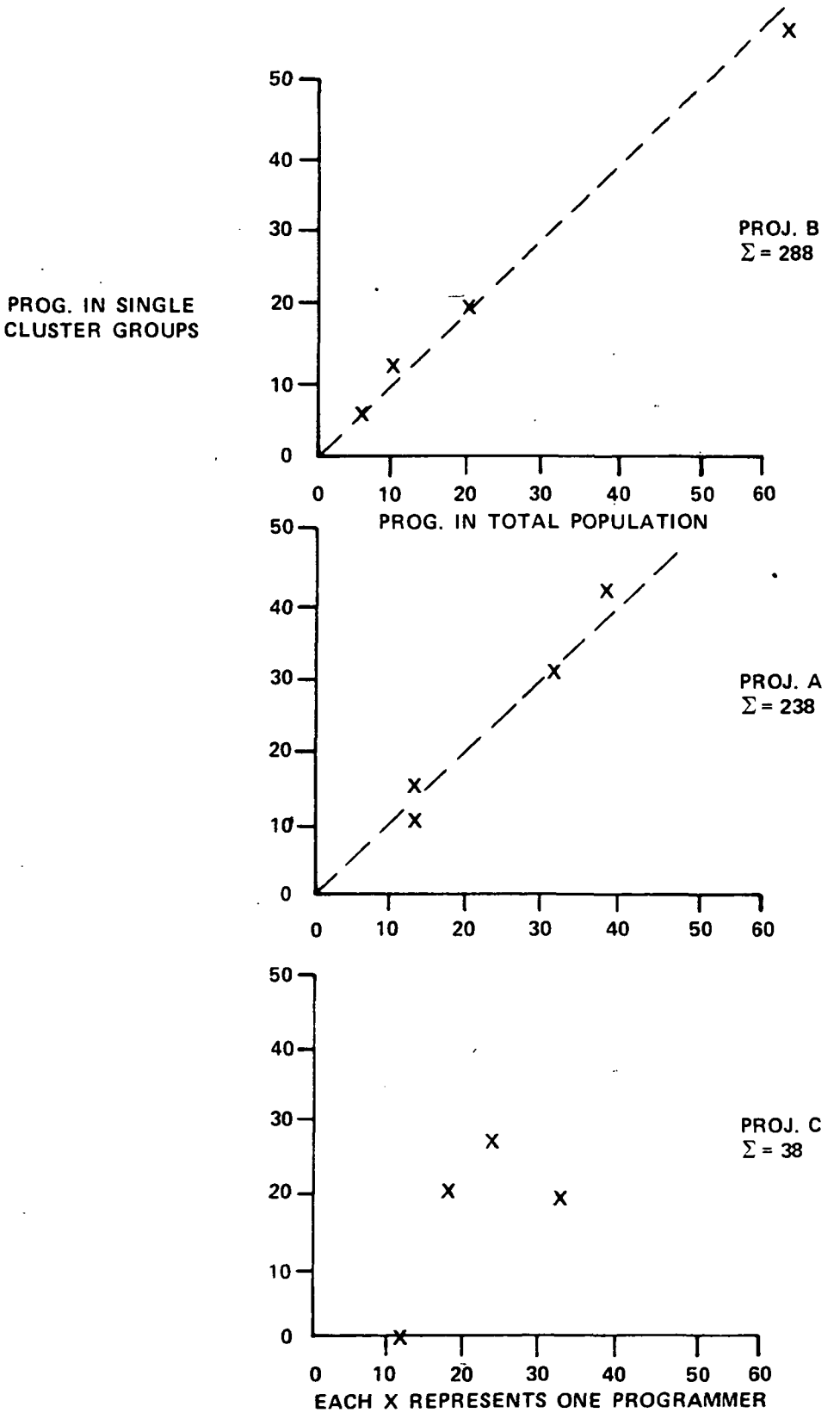
M. Zelkowitz
Univ. of Md.





A GIVEN PROGRAMMER APPEARING IN A GIVEN CLUSTER IS PROPORTIONAL TO CLUSTER SIZE – SHOWING EVEN DISTRIBUTION OF FORMS IN EACH CLUSTER

M. Zelkowitz
Univ. of Md.



FORMS THAT DO NOT CLUSTER ARE PROPORTIONAL TO NUMBER OF FORMS BY A GIVEN PROGRAMMER – SHOWING EVEN DISTRIBUTION

INVESTIGATIONS INTO SOFTWARE DEVELOPMENT IN THE SOFTWARE ENGINEERING LABORATORY

Victor R. Basili
Department of Computer Science
University of Maryland

The Software Engineering Laboratory (SEL) has been examining software on several projects with the goal of understanding the software development process and examining and refining existing models and metrics within the development environment. Early work has been predominantly in cost models in which the Putnam model for resource man-loading was examined. There were mixed results in that the model provided good estimation of development time given the maximum manning and budget estimates, but the Rayleigh curve did not fit the data well. This was at least in part due to a large amount of noise in the data. Several other curves, specifically a trapezoid, a parabola, and a straight line were used to fit the data. The trapezoid and parabola both fit approximately as well as the Rayleigh curve. Several techniques have been used in an attempt to smooth the data; the most effective so far uses modules as opposed to lines of code as the independent variable. Here we use the correlation between modules and lines of source code, and the invariant relationship discovered for the module handling rate across time to smooth the data.

A second effort has been to duplicate the relationships discovered by Walston/Felix concerning lines of code, effort in man weeks, staff size, module size, duration, etc. It was discovered that in the size range of the SEL projects, the relationship between lines of code and effort is almost linear (more linear than the Walston/Felix IBM data which deals with a larger range of projects). The productivity equation for the Walston/Felix data was $E = 5.2L^{.91}$ while the SEL data equation is $E = 1.35L^{.94}$. It is interesting to note, however, that the SEL data in general fits within one standard error of the IBM data. This result is highly plausible, since the majority of projects at NASA are of a similar nature–ground support software for satellites.

Current work is continuing in cost models by looking at one-formula paramaterized models, specifically the models of TRW, Doty Associates, and General Research Corporation. In each of these cases, a set of parameters is used to calculate a single value for resource effort. This effort is then allocated across the various phases and aspects of development. The goal is to normalize the parameters for the SEL environment. Also being examined are the man-loading models of Putnam and Parr. Parr's model is based on different assumptions than the model of Putnam. The results of the Parr model and the single-formula parameterized model will be compared to the Putnam model results.

Work has begun on the studies of effects on methodology and environment on software development. In each of the projects, a large list of subjective and objective measures, including the 29 parameters of Walston/Felix, are being estimated for the SEL data. An attempt is being made to distinguish the projects by these parameters. This will help in examining what effect each of these factors has on productivity and should provide valuable insight into the parameterization of the cost models.

The use of metrics, e.g., Halstead, McCabe, etc., and classification schemes for various modules, e.g., driver, input/output, etc., are being used to classify software and study the relationship between the metrics and classification schemes. This effort will further provide information for classifying software and provide insight into the cost models.

PANEL #2

DATA COLLECTION

P. Belford, Computer Sciences Corp.
M. Perie, FAA
L. Duvall, IITRI
P. de Feo, NASA/Ames

CENTRAL FLOW CONTROL SOFTWARE DEVELOPMENT: A CASE STUDY OF THE EFFECTIVENESS OF SOFTWARE ENGINEERING TECHNIQUES

Peter C. Belford and Richard A. Berg
Computer Sciences Corporation
Silver Spring, Maryland, U.S.A.

Thomas L. Hannan
Federal Aviation Administration
Washington, D.C., U.S.A.

Abstract

The purpose of this paper is to present cost and error data collected during the development cycle of a large-scale software effort, to analyze this data in comparison with other available data from similar projects, and to evaluate the effectiveness of the techniques utilized on the project. The project being reported on is Computer Sciences Corporation's development of the Central Flow Control Software System for the Federal Aviation Administration's Air Traffic Control System Command Center. Analysis of the cost data provides insight not only into the added development costs associated with severely limiting module sizes, but also into the effectiveness of various cost estimation techniques. The error data analysis supports the usefulness of the software engineering techniques which were used on the project in conjunction with definitive module-level test requirements. The paper provides a foundation upon which to establish the development and data collection environment for future software systems.

Introduction

Major software development projects employing controlled software engineering techniques occur infrequently over the life of an organization. It is even more uncommon for these controlled projects to have the management support required to collect sufficient data to evaluate the techniques utilized. In order that subsequent developers may have the opportunity to select effective software engineering techniques, more project details need to be collected, evaluated, and the results stored for their use.

This paper describes a major software development project, the software engineering practices employed, the data collection procedures and results obtained, and conclusions about the effectiveness of the practices as implemented on the project. The experiences gained from the project were not of the controlled, psychometric variety; the budget and deadlines were real, and the various lapses in data and, perhaps, in resolution, reflect these facts. Taking this into consideration, the following material is presented, not as conclusive proof of the efficacy of a particular methodology, but as experiences resulting from a representative large-scale development project.

Project and Approach

The Federal Aviation Administration (FAA) operates an Air Traffic Control System Command Center (ATCSCC) whose function is balancing the flow of air traffic so that in-flight delays are minimized. The Central Flow Control (CFC) System provides automation support for this function. A computer complex in Jacksonville, Florida, is linked with the major FAA nationwide facilities to provide up-to-date information about proposed flights and in-flight movements. This demand information is fed into a data base along with airport capacity information, and is subsequently used by ATCSCC personnel in an on-line query environment. The overall system can be described as an on-line (inquiry), real-time (flight data), transaction-oriented (independent asynchronous activities) information system.

The project was initiated in late 1975. It was decided to establish a rigid software architectural requirement and functional (stimulus/response) specification to ensure that the system would be able to evolve along with the application. Maintainability with an emphasis on modifiability[1] was the prime objective prior to contract initiation. However, a more recent grouping of these factors indicates that flexibility tended to become the primary goal, with maintainability second, and reliability a weak third for the initial system.[2]

Toward this end, the FAA specified the functional requirements,[3] utilizing an existing hardware configuration, and provided a baseline operating system. Computer Sciences Corporation (CSC) was competitively awarded a contract to modify the operating system, develop data base management and applications software, and provide support software for system development, generation, test, and performance evaluation. The award was made in April 1977, and the system was delivered in January 1979, six months prior to the operational readiness date of July 1979.

The selection of a development methodology was based as much on management criteria as on technical criteria. It was decided to move stepwise through the

development process, checkpointing each phase by baselining the output. With the requirements definition phase and the functional specification phase completed and their results baselined, the remaining development effort was allocated to three major phases: (1) the system design phase; (2) the unit design, code, and test phase; and (3) the system test phase. The final two phases were performed four separate times. Each time a portion of the software (called a Build) was implemented to demonstrate a subset of the functional capabilities of the system.

Software Engineering Methods

It became clear that successful execution of the development phase (within both budget and schedule) would require careful front-end attention paid to: (1) product definition, (2) tool utilization, (3) practice standardization, and (4) organizational structure. An attempt was made (with varying degrees of success) to define software and documentation products so that they would evolve naturally, and so that as much of these products as possible would be in machine-readable form. A set of tools and practices were selected to assist in those areas which had been troublesome in the past, most notably control and communication. Finally, an organization was formed based upon a Work Breakdown Structure (WBS), which was to serve as an accounting, data collection, and referencing mechanism as well as a basis for scheduling.

The system design phase involved the least infusion of modern practices and yielded the least amount of software engineering data. The products of the phase were (1) the Program Design Specification (PDS), (2) the System Development Plan, and (3) the System Test Plan. The PDS was developed using Hierarchy plus Input-Process-Output (H-IPO) diagrams. The Hierarchy (H-) diagrams eventually evolved into execution diagrams, which gained reasonable support, but the IPO diagrams, which met with some initial success, were quickly supplanted by Program Design Language (PDL). The PDL allowed for six basic structured constructs. Data was also addressed hierarchically in the PDS, and the CODASYL Data Description Language was employed.[4] This complemented the PDL, and both were updated for inclusion in the final documentation.

The unit design, code, and test phase was the most amenable to incorporation of modern practices and was the source of the most useful data. Additionally, the organization was adjusted during this phase to provide both a Quality Control group and an Independent Data Base Administrator.

Unit design was accomplished using PDL, and the unit test specification was generated by automated analysis of the PDL. This tool also verified the PDL for compliance with project quality standards. The test specifications were for unit testing based upon the decision-to-decision (DD) path structure of the design. The DD paths were determined from the constructs utilized in the PDL and were a relationship of the number of possible branch paths between constructs. The DD paths eventually demonstrated some highly advantageous properties (discussed subsequently in the section entitled "Presentation of Reduced Data"). Units, or modules, were constrained to single entry, single exit, and single function. They were documented at this stage by a machine-readable Prologue, which contained identification, operational-characteristic, cross-reference, data-definition, and processing-logic information.[5] Prologues were also automatically analyzed for completeness. Modules were subjected to Walkthroughs,[6] and the resultant error data, together with weekly resource expenditure sheets, the module PDL, Prologue, and Test Specifications were incorporated into individual Software Engineering Notebooks.

Unit coding and testing was based on the design in the PDL and Test Specifications. Structured code (in JOVIAL)[7] was derived from the PDL, and Test Procedures were developed based upon the Test Specifications; both of these items were then incorporated into the notebooks. Data interfaces were controlled by use of the JOVIAL data-description facility, COMPOOL, which was regulated by the Data Base Administrator. Resource utilization and error data were collected as they were in the previous phase; this data is presented and evaluated later in the paper.

The system test phase was carried out for each build by an Independent Test Team composed of both developer and user personnel. System Test Specifications and Procedures, in contrast to those at the module level, were for functional testing, and were traceable back to the requirements definition phase through the functional specifications. Errors were recorded on Test Team Trouble Reports, which were included in Build Test Reports.

Automated documentation tools were employed at the system level. The JOVIAL Automated Verification System (JAVS)[8] was modified for this project, and was used to produce program hierarchy (calling tree), program structure (DD paths), and cross-reference information, as well as to support the degree of test case coverage obtained. Intermediate JAVS outputs were also scanned by a specially developed software tool, which produced a Data Item Dictionary.

CFC Software Data Collection

Three general categories of data were collected on the CFC Project: activity data, software module structural data, and software error data. The activity data collected was man-hours expended by project personnel at the software subsystem level. Software module structural data included counts of the total number of executable source statements plus a count of the number of DD paths from the design for each module. Software error data included both walkthrough and execution time

errors. Walkthrough errors were recorded by quality control at the design walkthrough, and execution errors were recorded by the responsible programmers or the Independent Test Team, depending on the level of testing. Error data was collected both at the module (or unit) level and at the system level.

The procedures for data collection were based on the types of data and the mechanism used to collect each type. The data types and the corresponding basic collection devices were as follows:

- Activity data
  - Personnel: time accounting given at the subsystem level
- Module structural data
  - Physical structure: module source code
  - Logical structure: module PDL and test procedures
- Software error data
  - Module: programmer error log
  - System: test team trouble reports

Control of data collection was maintained within the development departments and monitored by the quality control staff on a continuing basis. Organization and delivery of the final data package was performed by the quality control staff. The following subsections describe in detail the forms and procedures used to collect and summarize the three data types.

Activity Data Collection

Time data in man-hours was collected for all personnel on the project at the functional subsystem level. This time data was collected by means of a CFC Project Data Collection Form that was distributed weekly and filled out along with weekly time cards by each person on the project. The number of man-hours expended in each subsystem for design, code, test, library maintenance, throwaway code development, management, and quality assurance functions performed within the subsystem was compiled from this data.

The data was collected for each of the four builds of the system. Within each build, data was presented for the Application/Simulation (APS) Subsystem, the Data Assembler (DA) Subsystem, the Data Base (DB) Subsystem, and the Data Reduction and Analysis (RA) Subsystem. These subsystems represented the major functions performed by the CFC System that were programmed in JOVIAL. The time spent in functional design and system level testing was not included in the build man-hour results.

Software Structural Data Collection

Two types of software structural data were obtained from the CFC Development Project. The data was acquired from examination of the module source code and PDL. The module source code data was an estimate of the physical size of a module measured by a count of the total number of executable source statements. The PDL data defined the logical structure of the design of the module, obtained by a count of the total number of DD paths in the design. Another measure of logical structure utilized was a count of the number of test cases used to exercise all the DD paths identified within a module during the design phase. The test case data was obtained from the test procedures in the Software Engineering Notebooks.

Software Error Collection

Software error data on the CFC Project was collected during the design and testing phases. Errors detected in the design phase were measured by the total number of design errors discovered during the design walkthrough of a given module. These error counts were collected by the quality control staff during each walkthrough.

Errors encountered during module or unit level testing were collected by the responsible programmer and summarized for each build. System level errors were collected by the Independent Test Team for each unsuccessful run. The failure information was derived from an analysis of the program output. If a failure was caused by more than one error, all errors were listed. Errors were also recorded during system acceptance testing. While not on a build basis, these errors provided information about problems encountered during the integration of the final system.

Presentation of the Data

This section presents the raw data collected on the CFC Project. Data is presented in the three categories described in the preceding section, and is presented for every build in which it was available.

Activity data is shown in Tables 1 through 4. Tables 1 through 3 show the man-hours spent per subsystem in the activity categories of detailed design, code, unit test, library maintenance, throwaway code development, and management and technical direction by task leaders in the subsystem plus quality control functions performed by subsystem personnel. Data is presented for Builds 2, 3, and 4 of the system. Build 1 of the system is not presented since it occurred at a time prior to the institution of the reporting mechanism. However, total man-hours statistics are available for Build 1. Table 4 presents the total man-hours spent per subsystem for each of the four builds of the system. Tables 5 through 7 show the number of executable lines of code, the number of DD paths, and the number of test cases,

TABLE 1. BUILD 2 MAN-HOURS IN REPORTED ACTIVITY CATEGORIES

CATEGORY	APS SUBSYSTEM	DA SUBSYSTEM	DB SUBSYSTEM	RA SUBSYSTEM	TOTALS
Design	1293	885	462	0	2640
Code	731	432	362	0	1525
Test	1114	834	711	0	2659
Library Maint.	315	111	122	0	548
Throwaway Code	77	121	198	0	396
Other*	564	346	175	0	885
TOTALS	4094	2729	2030	0	8853

TABLE 2. BUILD 3 MAN-HOURS IN REPORTED ACTIVITY CATEGORIES

CATEGORY	APS SUBSYSTEM	DA SUBSYSTEM	DB SUBSYSTEM	RA SUBSYSTEM	TOTALS
Design	1937	775	378	1557	5247
Code	717	381	414	591	2303
Test	1215	985	591	1198	4080
Library Maint.	192	118	24	83	417
Throwaway Code	96	113	65	145	419
Other*	913	165	309	414	1801
TOTALS	5171	2537	1981	3979	14667

TABLE 3. BUILD 4 MAN-HOURS IN REPORTED ACTIVITY CATEGORIES

CATEGORY	APS SUBSYSTEM	DA SUBSYSTEM	DB SUBSYSTEM	RA SUBSYSTEM	TOTALS
Design	416	154	753	1085	2408
Code	170	196	498	397	1261
Test	763	957	201	529	2450
Library Maint.	489	199	109	24	821
Throwaway Code	60	9	7	85	161
Other*	861	172	522	247	1802
TOTALS	2779	1387	2090	2637	8893

TABLE 4. MAN-HOURS EXPENDED BY SUBSYSTEM FOR EACH BUILD

BUILD	APS SUBSYSTEM	DA SUBSYSTEM	DB SUBSYSTEM	RA SUBSYSTEM	SUBTOTAL
1	1480	1843	898	0	5221
2	4094	2729	2030	0	8853
3	5171	2537	1981	3979	14567
4	2779	1387	2090	2637	8893
TOTAL	13524	8996	7399	6615	36034

TABLE 5. EXECUTABLE LINES OF CODE PER SUBSYSTEM PER BUILD

BUILD	APS	DA	DB	RA	TOTAL
1	760	1357	840	0	3257
2	2759	1656	922	0	5337
3	2360	2933	2061	3035	10389
4	1503	939	1153	1364	4959
TOTAL	7482	7185	4676	4399	23742

TABLE 6. DD PATHS PER SUBSYSTEM PER BUILD

BUILD	APS	DA	DB	RA	TOTAL
1	233	530	146	0	909
2	987	403	250	0	1640
3	799	543	503	1068	2915
4	435	154	323	343	1255
TOTAL	2454	1632	1222	1411	6719

TABLE 7. TEST CASES PER SUBSYSTEM PER BUILD

BUILD	APS	DA	DB	RA	TOTAL
1	119	170	76	0	365
2	344	135	173	0	652
3	283	127	134	197	741
4	169	26	139	88	422
TOTAL	915	558	482	385	2340

*Includes management and technical direction by subsystem leader and quality assurance functions performed by subsystem personnel.

P. Belford
CSC
4 of 20

respectively, for each of the four system builds. Table 3 gives the error statistics collected for Builds 2 and 3 of the system and the total number of software errors detected during acceptance testing. Build 1 was completed before error reporting mechanisms were in place, and Build 4 results were not available.

TABLE 3. ERROR STATISTICS FOR THE CENTRAL FLOW CONTROL SYSTEM

BUILD	DESIGN WALKTHROUGH	MODULE* LEVEL TESTING	SYSTEM LEVEL TESTING
2	44	290	18
3	63	223	20
Acceptance Testing	N/A	N/A	21

*APS and DA Subsystems only.

Presentation of Reduced Data

The purpose of this section is to analyze the data presented in the preceding section. This analysis attempts to provide quantitative evaluation of the effectiveness of the software engineering techniques utilized on the CFC Project. Since the project has been completed and accepted by the FAA, this analysis evaluates the final results of the project.

The first analysis performed concerned the relationship of the sizes of software entities compared to the cost of their development. The size of a software entity was measured in terms of both the number of lines of executable source code and the number of decision paths (DD paths) in the design. Cost was always measured in man-hours expended.

Module Level and Build Level were the two software entities evaluated. At the module level, Figure 1 presents a plot of the number of man-hours expended versus the number of lines of executable code for each of a randomly selected set of 50 modules. Any conclusive trend is not at all obvious by analysis of the curve. However, since the true comparison of developmental costs is in terms of the number of lines of code produced per man-hour expended, further evaluation was performed.

The data, therefore, was evaluated in terms of the number of lines of code developed per man-hour (a "relative" measure of cost) as a function of the number of lines of code in the module (Figure 2). If a curve could be formed from the data, the optimal module size would be the highest point on the curve (i.e., the module size for which the maximum number of lines of code would be developed in each man-hour expended).

The data presented in Figure 2 shows that module sizes of between 0 and 40 lines of executable source code never (in fact, without exception in this sample) produce a productivity of more than one line of code produced per man-hour expended. However, for modules of greater

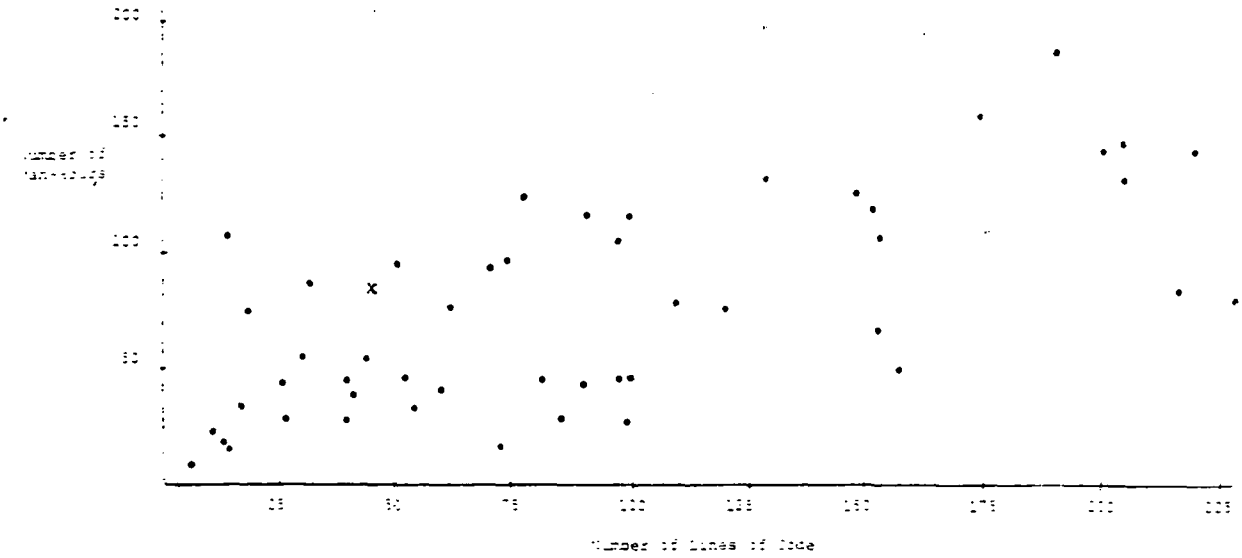


FIGURE 1. CORRELATION OF MAN-HOURS TO MODULE SIZES

P. Belford
CSC
5 of 20

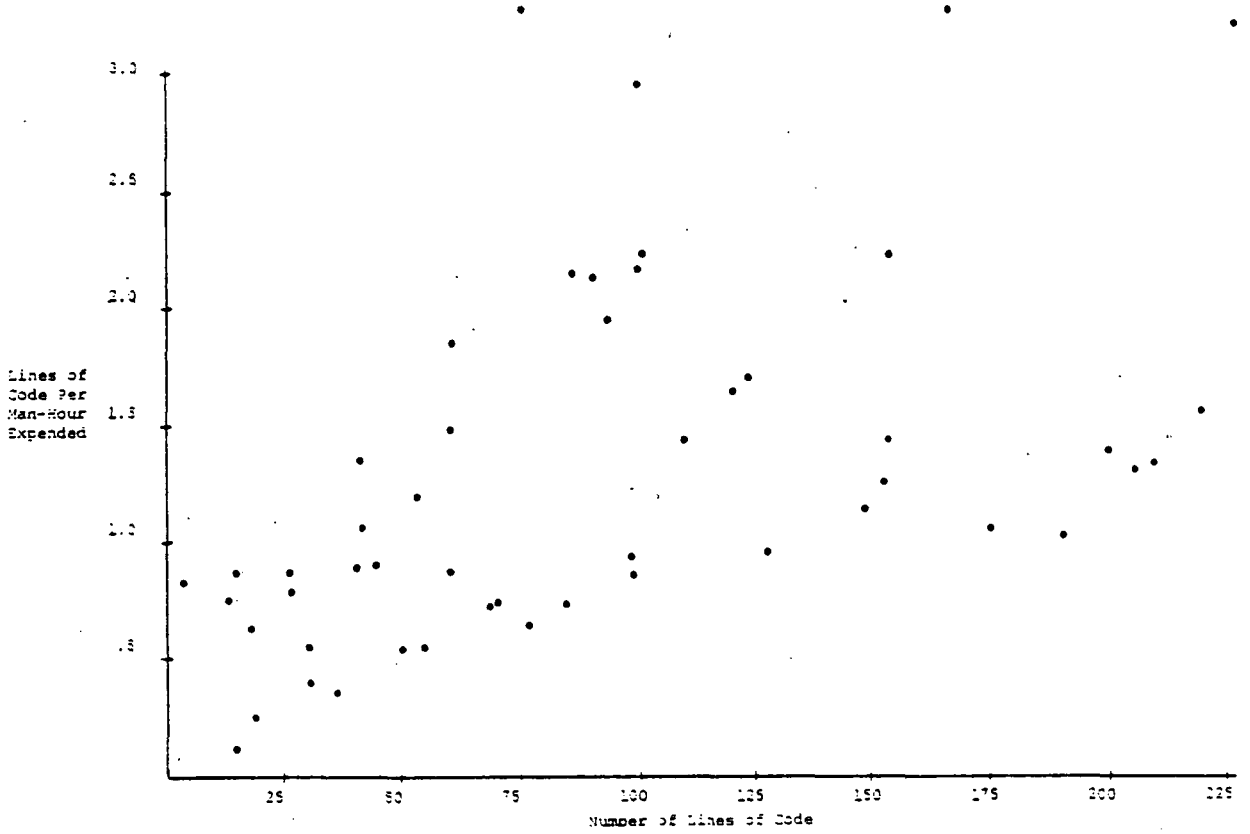


FIGURE 2. CORRELATION OF LINES OF CODE DEVELOPED PER MAN-HOUR COMPARED TO MODULE SIZE

than 40 lines of executable source code (greater than 100 without exception), the productivity is generally greater than one line of code produced per man-hour expended.

Although the data presented only reflects development costs and not operational and maintenance costs, the general theory of restricting modules to 50 lines of executable source code or less does not appear to be cost effective when considering developmental costs only. If the "single entry, single exit, single function" rule is strictly adhered to, the module size should not be utilized as a standard. In contrast, it appears that modules of 40 lines of code or greater should be encouraged. It is important to note that the CFC modules, regardless of size, followed the "single entry, single exit, single function" concept of module definition.

In order to support these conclusions, a measure of module complexity, number of DD paths in the design, was also compared to developmental costs. The same 50 modules were utilized. This time, the relative cost in number of DD paths generated per man-hour was plotted against the complexity in number of DD paths. Figure 3 shows the results of this analysis. Again, the more complex the module, the lower the relative developmental costs.

These cost analyses seem to point out that within the "single entry, single exit, single function" concept, the larger the module, the lower the per-unit-of-size developmental costs.

The results of this analysis seem to indicate that restrictions limiting module size should not be a driving factor. Single-function modules of 100 or even 200 lines of executable source code should be acceptable.

The relative size of a build was also analyzed. Figure 4 shows the relationship between the size of a build in number of lines of executable source code per subsystem and the number of man-hours expended against that subsystem in a build. This plot shows an obvious correlation of size to cost. With the single exception of the DA Subsystem for Build 3, which was accomplished on third shift (the implications of which will not be discussed here), the cost-to-size relationship is linear. Hence, if the single-function module concept is controlled, the size of a build appears to have no impact on the relative cost of producing that build.

Another analysis was performed to evaluate the relationship between actual development costs and cost estimation techniques. The number of man-hours expended

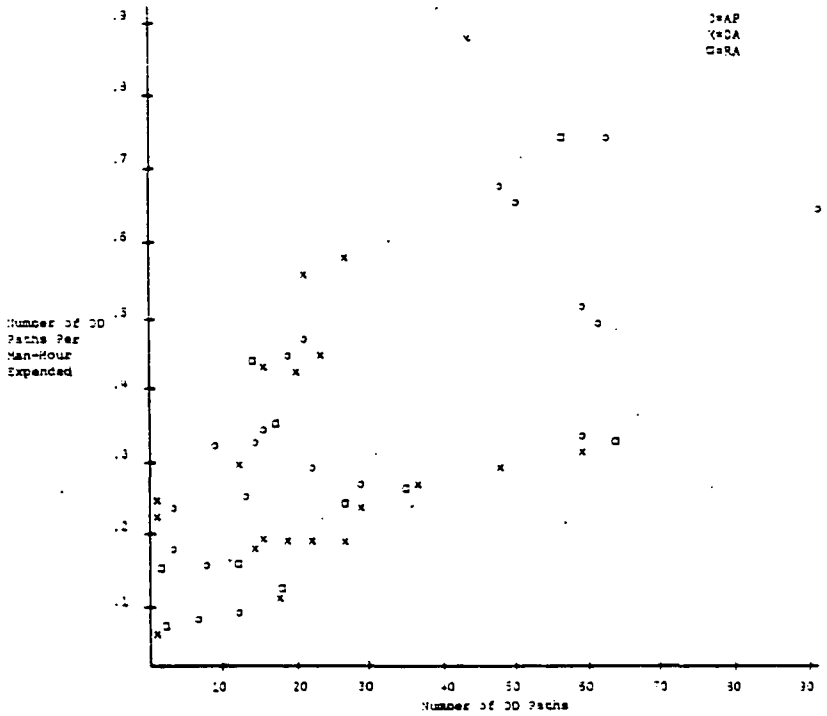


FIGURE 3. CORRELATION OF NUMBER OF DD PATHS PER MAN-HOUR TO MODULE COMPLEXITY

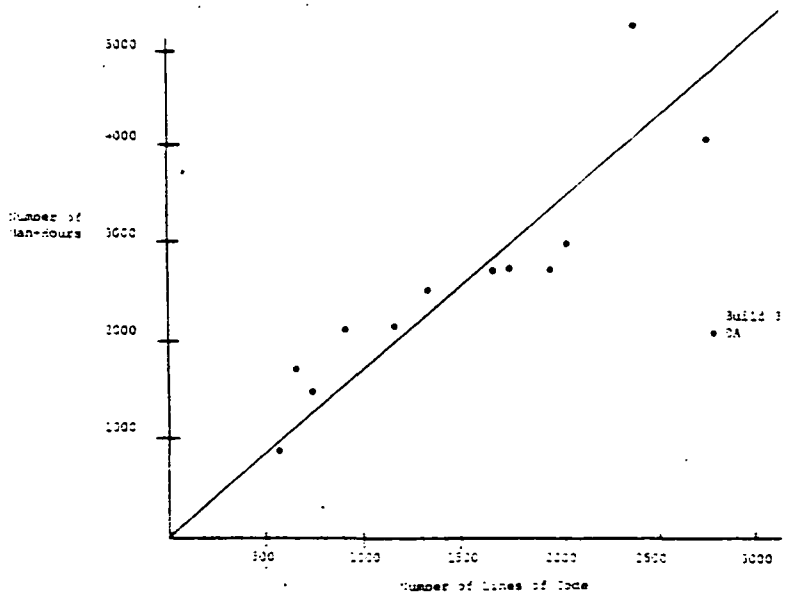


FIGURE 4. CORRELATION OF MAN-HOURS TO BUILD SIZE

per line of executable source code was compared to the number of man-hours expended per DD path in the design and to the number of man-hours expended per test case (see Table 9). This comparison was accomplished using a coefficient of variation (i.e., the ratio of the standard deviation to the mean).

TABLE 9. COSTING PARAMETER COMPARISONS

Build/Subsystem	Man-Hours/Lines Code	Man-Hours/DD Path	Man-Hours/Test Case
Build 1 APS	1.35	6.35	12.44
Build 1 CB	1.56	6.15	11.92
Build 1 CA	1.45	5.36	10.53
Build 2 APS	1.48	6.15	11.90
Build 2 CB	1.20	5.12	27.81
Build 2 CA	1.35	5.77	20.21
Build 3 APS	1.19	5.07	18.27
Build 3 CB	1.40	5.73	14.85
Build 3 CA	.86	4.66	13.98
Build 3 PA	1.31	5.72	13.29
Build 4 APS	1.73	6.39	15.44
Build 4 CB	1.81	6.47	15.34
Build 4 CA	2.25	12.25	32.58
Build 4 PA	1.93	7.69	29.97
COV	35.13%	32.13%	75.53%

The statistical correlation was not diverse enough to support differentiation between DD paths in the design and lines of code in terms of total cost estimation techniques. Number of man-hours per test case was shown not to be a viable cost estimation technique due to its high coefficient of variation (COV). Therefore, a separate analysis was performed to determine a better costing parameter that could be utilized at each of the detail design, coding, and testing phases of development. Since the known quantity at the completion of each phase is DD paths in the detail design phase, lines of code in the coding phase, and test cases in the testing phase, this information could be utilized to refine original cost estimates as a project progresses. The data presented on costing provides enough information to support development of initial cost estimation algorithms based on actual development products (e.g., DD paths, lines of code, and test cases).

An analysis was performed on these cost estimations utilizing the data from the APS Subsystem for all four builds. Three cost estimation algorithms were developed, one for each development phase. The basis for these algorithms is the data previously presented in Table 9. The "Lines of Code" algorithm utilizes 1.34 times the number of lines of executable source code to yield the estimated number of man-hours. The "DD-Path" algorithm utilizes 5.34 times the number of DD paths in the design to yield the estimated number of man-hours. The "Test Case" algorithm utilizes 14.76 times the number of test cases to be performed. All three algorithms were then applied to the other subsystems. These results are presented in Table 10.

The low "average percentage deviation" of the DD-path algorithm shows that the number of DD paths provides an accurate prediction of the total man-hours required. This technique provides the added advantage of supporting periodic updating of the estimation as the actual number of DD paths is finalized. If PDL is used, this variable is known early (i.e., at the completion of the design phase).

The effectiveness of the software engineering techniques utilized was also analyzed. The true effectiveness of the software engineering techniques can best be measured by the reliability and maintainability of the product. Although data is not yet available to support definitive reliability and maintainability measures of the CFC System, error rate data was available within CFC and was used to estimate the effectiveness of the software engineering techniques employed on the project.

In order to evaluate the CFC error rate with some defined industry averages, the Rome Air Development Center (RADC) Software Reliability Study[9] was utilized. This report presents the error rate of two JOVIAL projects at system level testing. This error rate worked out to be about 1 error in every 35 lines of code. The CFC error rate, calculated from Tables 5 and 9, shows 1 error in every 28 lines of code, detected at the module level. At the system level testing of CFC, an order of magnitude fewer number of errors (i.e., 1 error for every 382 lines of code) than at the module level were found. During final acceptance level testing, a 3-month user/customer-conducted testing phase, still fewer errors were found. In the 23,742 lines of executable code discussed in this paper, only 21 software errors were detected. This is an error rate of 1 error in every 1,131 lines of code.

The error rate implies two conclusions about the development approach. First, the software engineering techniques utilized were very effective. The CFC error detection rate comparable to that reported in the RADC study was noticed an entire development phase earlier. More errors were found earlier, presumably leaving fewer errors in the final system. This should result in a significant cost savings, since the cost to correct an error increases the longer it remains in the code. Second, the testing approach proved to be quite effective. The quantification of module level testing requirements, specifying that all DD paths in the design must be exercised at the module level, provided significant advantages over the traditional testing approach carried on by the programmers. This concept proved to exercise 98 percent of all the decision paths in the code. Additionally, within one subsystem where these unique module level test specifications were rigidly applied, the acceptance testing error rate was only 1 error detected in every 1,371 lines of executable code; whereas within a subsystem where module level test specifications were loosely applied, the acceptance error rate was 1 error detected in every 733 lines of code. This roughly implies an overall effectiveness increase of over 100 percent

per line of executable source code was compared to the number of man-hours expended per DD path in the design and to the number of man-hours expended per test case (see Table 9). This comparison was accomplished using a coefficient of variation (i.e., the ratio of the standard deviation to the mean).

TABLE 9. COSTING PARAMETER COMPARISONS

Build/Subsystem	Man-Hours/Lines Code	Man-Hours/DD Path	Man-Hours/Test Case
Build 1 APS	1.35	5.35	12.40
Build 1 CB	1.66	6.15	11.12
Build 1 DA	1.45	5.38	13.53
Build 2 APS	1.48	4.15	11.90
Build 2 CB	1.20	3.12	17.81
Build 2 DA	1.65	6.77	20.22
Build 3 APS	1.13	4.47	16.17
Build 3 CB	1.40	5.73	14.85
Build 3 DA	.98	4.66	19.38
Build 3 RA	1.31	3.72	13.19
Build 4 APS	1.73	6.39	16.40
Build 4 CB	1.81	6.47	15.36
Build 4 DA	1.35	12.25	22.58
Build 4 RA	1.33	7.69	19.37
COV	15.13%	32.13%	21.53%

The statistical correlation was not diverse enough to support differentiation between DD paths in the design and lines of code in terms of total cost estimation techniques. Number of man-hours per test case was shown not to be a viable cost estimation technique due to its high coefficient of variation (COV). Therefore, a separate analysis was performed to determine a better costing parameter that could be utilized at each of the detail design, coding, and testing phases of development. Since the known quantity at the completion of each phase is DD paths in the detail design phase, lines of code in the coding phase, and test cases in the testing phase, this information could be utilized to refine original cost estimates as a project progresses. The data presented on costing provides enough information to support development of initial cost estimation algorithms based on actual development products (e.g., DD paths, lines of code, and test cases).

An analysis was performed on these cost estimations utilizing the data from the APS Subsystem for all four builds. Three cost estimation algorithms were developed, one for each development phase. The basis for these algorithms is the data previously presented in Table 9. The "Lines of Code" algorithm utilizes 1.34 times the number of lines of executable source code to yield the estimated number of man-hours. The "DD-Path" algorithm utilizes 5.34 times the number of DD paths in the design to yield the estimated number of man-hours. The "Test Case" algorithm utilizes 14.76 times the number of test cases to be performed. All three algorithms were then applied to the other subsystems.. These results are presented in Table 10.

The low "average percentage deviation" of the DD-path algorithm shows that the number of DD paths provides an accurate prediction of the total man-hours required. This technique provides the added advantage of supporting periodic updating of the estimation as the actual number of DD paths is finalized. If PDL is used, this variable is known early (i.e., at the completion of the design phase).

The effectiveness of the software engineering techniques utilized was also analyzed. The true effectiveness of the software engineering techniques can best be measured by the reliability and maintainability of the product. Although data is not yet available to support definitive reliability and maintainability measures of the CFC System, error rate data was available within CFC and was used to estimate the effectiveness of the software engineering techniques employed on the project.

In order to evaluate the CFC error rate with some defined industry averages, the Rome Air Development Center (RADC) Software Reliability Study[9] was utilized. This report presents the error rate of two JOVIAL projects at system level testing. This error rate worked out to be about 1 error in every 35 lines of code. The CFC error rate, calculated from Tables 5 and 8, shows 1 error in every 28 lines of code, detected at the module level. At the system level testing of CFC, an order of magnitude fewer number of errors (i.e., 1 error for every 382 lines of code) than at the module level were found. During final acceptance level testing, a 3-month user/customer-conducted testing phase, still fewer errors were found. In the 23,742 lines of executable code discussed in this paper, only 21 software errors were detected. This is an error rate of 1 error in every 1,131 lines of code.

The error rate implies two conclusions about the development approach. First, the software engineering techniques utilized were very effective. The CFC error detection rate comparable to that reported in the RADC study was noticed an entire development phase earlier. More errors were found earlier, presumably leaving fewer errors in the final system. This should result in a significant cost savings, since the cost to correct an error increases the longer it remains in the code. Second, the testing approach proved to be quite effective. The quantification of module level testing requirements, specifying that all DD paths in the design must be exercised at the module level, provided significant advantages over the traditional testing approach carried on by the programmers. This concept proved to exercise 98 percent of all the decision paths in the code. Additionally, within one subsystem where these unique module level test specifications were rigidly applied, the acceptance testing error rate was only 1 error detected in every 1,371 lines of executable code; whereas within a subsystem where module level test specifications were loosely applied, the acceptance error rate was 1 error detected in every 733 lines of code. This roughly implies an overall effectiveness increase of over 100 percent

TABLE 10. COST ESTIMATION COMPARISONS

		"Lines of Code" Algorithm		"DD Path" Algorithm		"Test Case" Algorithm	
	Actual Man-Hours	Estimated Man-Hours	Deviation (%)	Estimated Man-Hours	Deviation (%)	Estimated Man-Hours	Deviation (%)
Build 1 DB	898	994	10.69	953	6.01	1122	24.94
Build 1 DA	2843	3601	26.66	3095	8.86	3985	40.17
Build 2 DB	2030	1696	16.45	1460	28.08	1077	46.95
Build 2 DA	2729	3047	11.65	2354	13.54	1993	26.97
Build 3 DB	2861	3792	31.62	2938	1.98	2863	.62
Build 3 DA	2537	5397	112.73	3193	25.86	1975	26.09
Build 3 RA	3979	5584	40.37	6237	56.79	4384	10.21
Build 4 DB	2090	2122	1.53	1886	9.76	2052	1.82
Build 4 DA	1887	1176	37.68	899	52.36	384	79.65
Build 4 RA	2637	2510	4.82	2003	24.04	1299	50.74
Average Deviation			29.42		22.61		30.32

attributable to using these module level test specifications.

Summary

In conclusion, the authors feel that a significant baseline has been established in the quantification of software engineering techniques. The success of the CFC project, together with the supporting data that was collected, provides a foundation upon which to build successor systems. The key factors to be considered in setting up these future system programs is to understand the development environment, to collect data during the development effort to support the upgrading of projections, and to support the periodic evaluation of the data to provide insight into the product. This should provide sufficient visibility to keep a project out of trouble, while at the same time supporting evolution of more effective project plans.

References

1. Boehm, B. W., et al., Characteristics of Software Quality, TRW Systems Group, TRW-SS-73-09, December 1973.

2. McCall, J. A., et al., Factors in Software Quality, Final Technical Report (3 vols.), Rome Air Development Center, RADC-TR-77-369, November 1977.

3. Central Flow Control Computer Program Specifications, Final Report (5 vols.), Federal Aviation Administration, FAA-RD-76-157, September 1976.

4. CODASYL Data Base Task Group Report, Conference on Data Systems Languages, April 1971.

5. Central Flow Control Quality Assurance Plan, Final Report, Computer Sciences Corporation, CSC/SD-78/6060, April 1978.

6. Fagan, M. E., "Design and Code Inspections to Reduce Errors in Program Development," IBM Systems Journal, Vol. 15, No. 3, 1976.

7. NAS Operational Support System, IBM 9020 JOVIAL Language Manual, NASP-9288-02, May 1975.

8. Gannon, C., et al., JAVS - JOVIAL Automated Verification System, JAVS Technical Report (3 vols.), General Research Corporation, CR-1-722/1, June 1978.

9. Software Reliability Study, Rome Air Development Center, RADC-TR-74-250, October 1974.

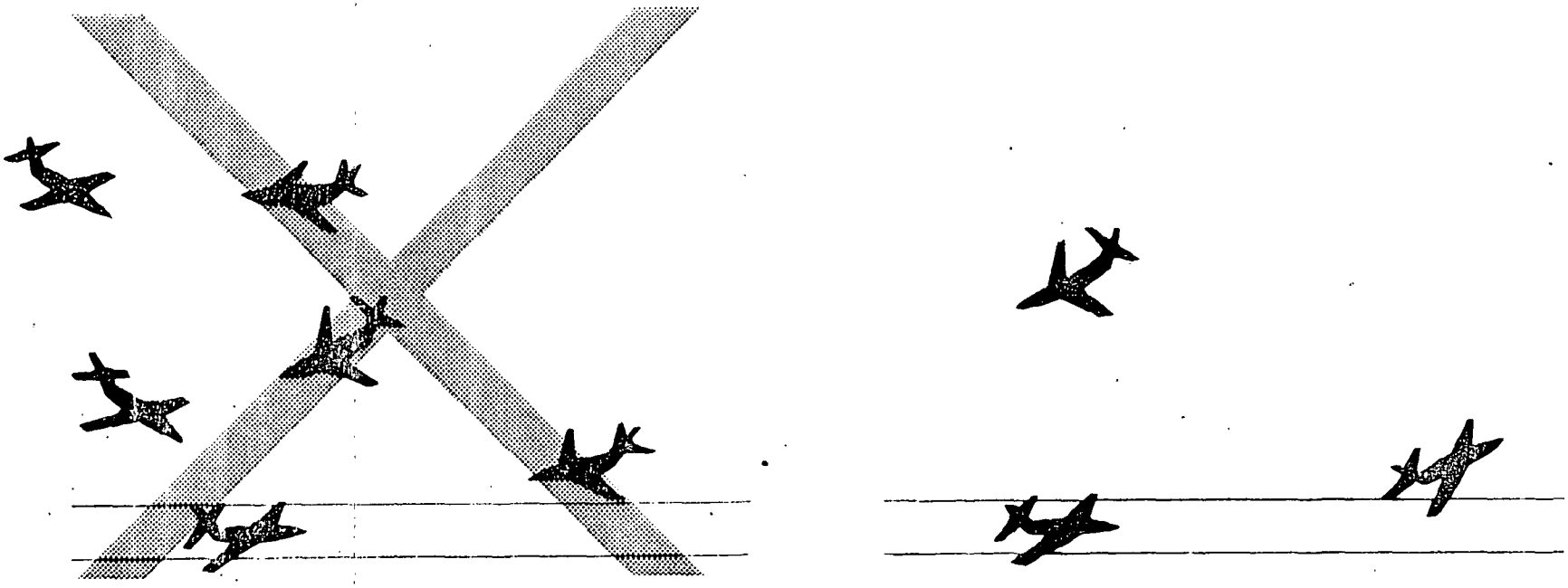
CENTRAL FLOW CONTROL

T. L. HANNAN, FEDERAL AVIATION ADMINISTRATION

R. A. BERG, COMPUTER SCIENCES CORPORATION

BEL-0-79

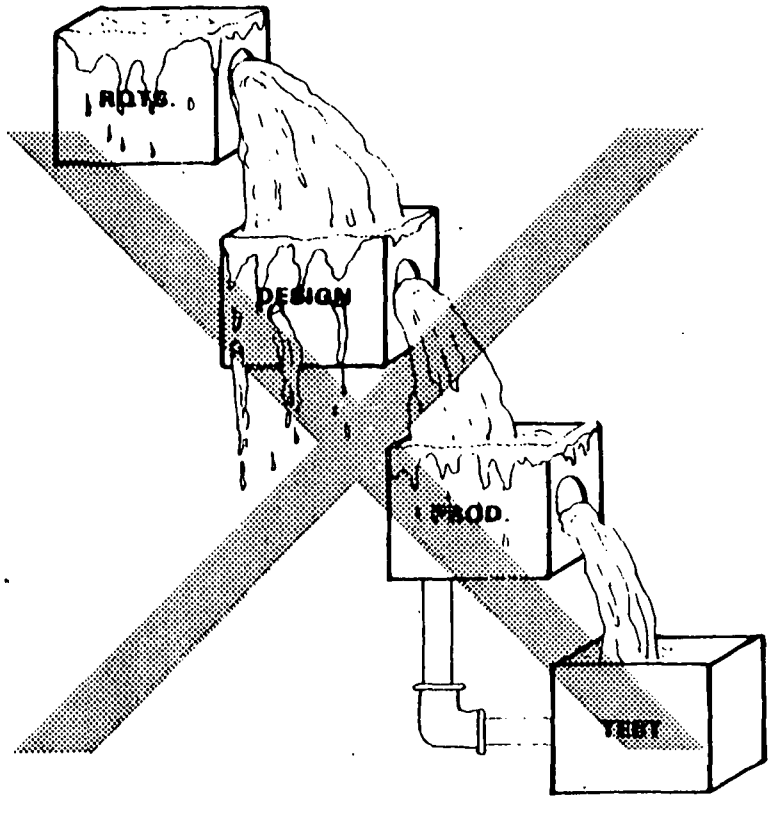
CENTRAL FLOW CONTROL FUNCTIONAL PURPOSE

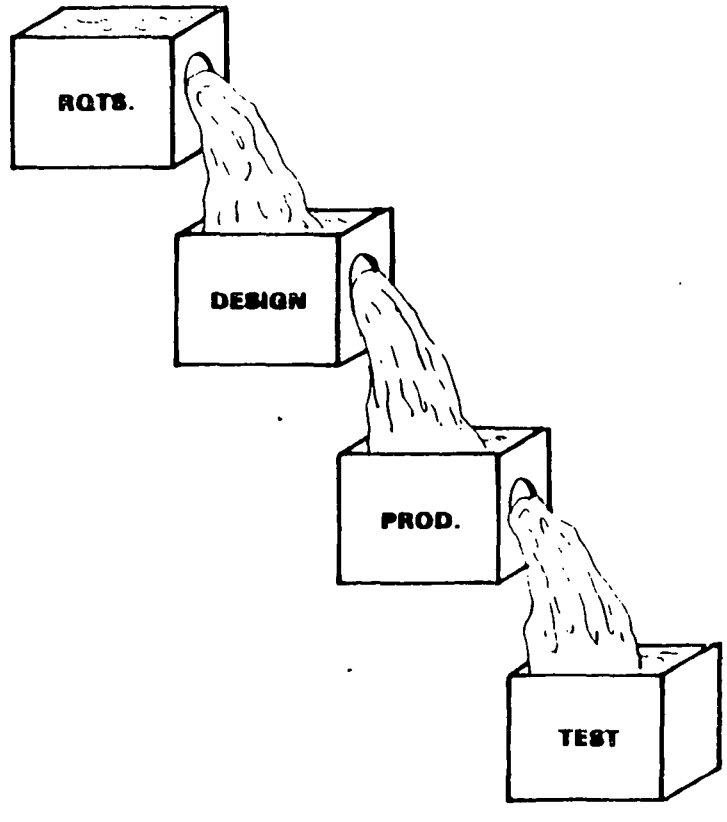


- LONG RANGE TRAFFIC OVERLOAD PREDICTION
- REDUCTION IN AIR TRAFFIC DELAYS
- FUEL SAVINGS

BEL-8-78

CENTRAL FLOW CONTROL
SOFTWARE ENGINEERING PURPOSE





- PLANNED APPROACH
- STRUCTURED TESTING
- COMPREHENSIVE DATA COLLECTION

SEL-8-78

P. Belford
CSC
13 of 20

DATA ANALYSIS AREAS AND RESULTS

- MODULE SIZE ANALYSIS
- BUILD SIZE ANALYSIS
- COSTING METHODOLOGY RESULTS
- TESTING METHODOLOGY RESULTS

BEL-8-78

DATA COLLECTED

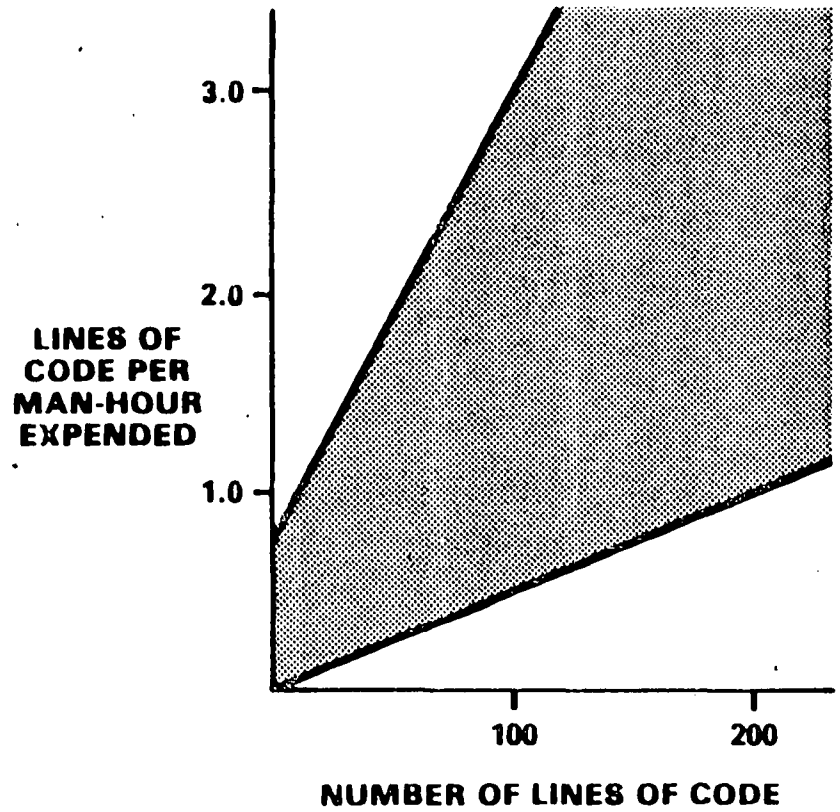
- **ACTIVITY DATA**
  - **PERSONNEL: TIME ACCOUNTING GIVEN AT THE SUBSYSTEM LEVEL**
- **MODULE STRUCTURE DATA**
  - **PHYSICAL STRUCTURE: MODULE SOURCE CODE**
  - **LOGICAL STRUCTURE: MODULE PDL AND TEST SPECIFICATIONS**
- **SOFTWARE ERROR DATA**
  - **MODULE: PROGRAMMER ERROR LOG**
  - **SYSTEM: TEST TEAM TROUBLE REPORTS**

DATA PRESENTED

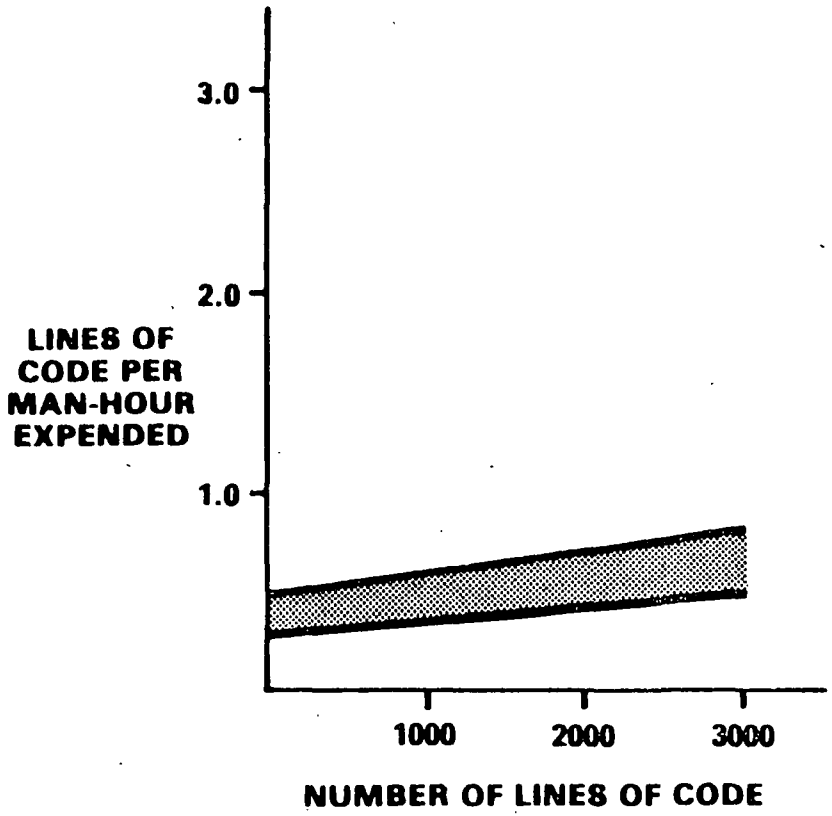
- DATA PRESENTED ON THE MAJOR SUBSYSTEMS OF CFC THAT WERE CODED IN JOVIAL
- 38034 MAN-HOURS OF DIRECT LABOR AND 23742 LINES OF EXECUTABLE CODE REPRESENTED OVERALL
- OVERALL DATA PRESENTED:

  MAN-HOURS, EXECUTABLE LINES OF CODE, DD-PATHS, TEST CASES, ERRORS

OVER LIMITING SIZE
NOT COST EFFECTIVE



MODULE SIZE



BUILD SIZE

COSTING METHODOLOGY

- MAN-HOURS PER LINE OF CODE/DD PATH/TEST CASE WERE CALCULATED FOR EACH SUBSYSTEM FOR EACH BUILD

- RESULTS:

  COV (LINE OF CODE) = 35%
  COV (DD PATH) = 32%
  COV (TEST CASE) = 76% (ELIMINATED AS USEFUL ESTIMATOR)

- APS SUBSYSTEM USE AS ESTIMATOR FOR OTHER SUBSYSTEMS AND RESULTS COMPARED WITH ACTUALS

- RESULTS:

  AVERAGE DEVIATION (LINES OF CODE) = 24%
  (DD PATHS) = 22%

- CONCLUSION:

  DD PATHS CAN BE USEFUL IN REFINING INITIAL COST ESTIMATES

TESTING METHODOLOGY

- DISCIPLINED APPROACH
- TEST SPECS GENERATED FROM PDL
- TESTS EXERCISE ALL DD PATHS IN DESIGN
- VERIFIED AT DESIGN WALKTHROUGHS

RESULTS

- 1 ERROR PER 28 LINES OF CODE AT MODULE LEVEL
- 1 ERROR 382 LINES OF CODE AT SYSTEM LEVEL
- 1 ERROR PER 1131 LINES OF CODE AT ACCEPTANCE LEVEL

NOTE: SUBSYSTEM RIGIDLY FOLLOWING PDL/TESTING STANDARDS
1 ERROR PER 1871 LINES OF CODE DETECTION AND CORRECTION TIME AVERAGED 8 PERSON HOURS.

SUBSYSTEM LOOSELY FOLLOWING DL/TESTING STANDARDS
1 ERROR PER 733 LINES OF CODE DETECTION AND CORRECTION TIME AVERAGED 40 PERSON HOURS.

BEL-9-79

SUMMARY

- BUILD SIZE HAS LITTLE COST IMPACT
- OVER RESTRICTING MODULE SIZE IS NOT COST EFFECTIVE IN THE INITIAL DEVELOPMENT
- RIGID ADHERENCE TO THE METHODOLOGY CAN REDUCE COST
- THE METHODOLOGY DID NOT SIGNIFICANTLY REDUCE THE NUMBER OF ERRORS BUT DID ALLOW EARLY DETECTION OF MOST ERRORS
- THE NUMBER OF DD PATHS IS DIRECTLY RELATED TO LINES OF EXECUTABLE CODE AND CAN BE USED TO REFINE COST ESTIMATES AFTER THE DESIGN STAGE

BEL-9-79

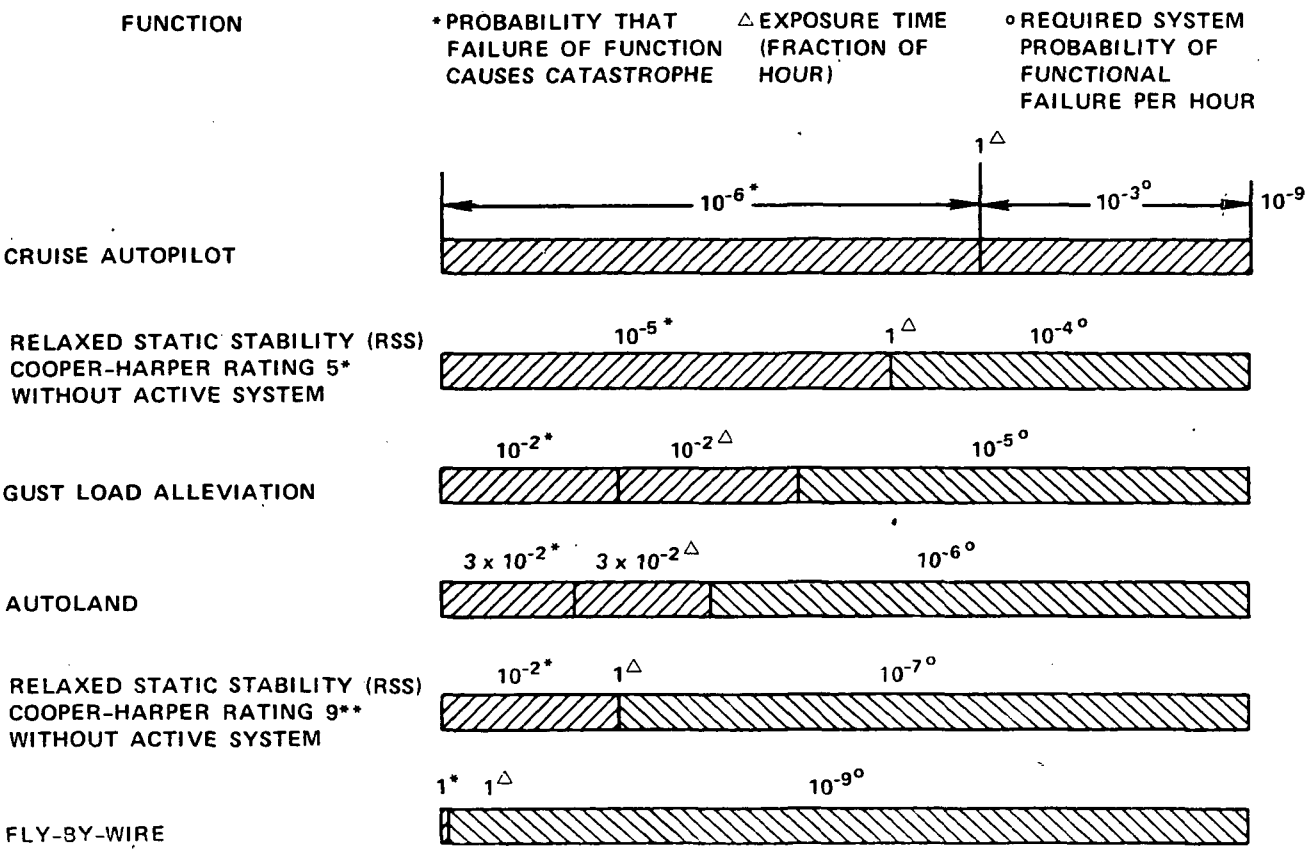
VALIDATION TECHNOLOGY

JOINT NASA/FAA PROGRAM

RTOP 512-54-01

M. Perie
FAA

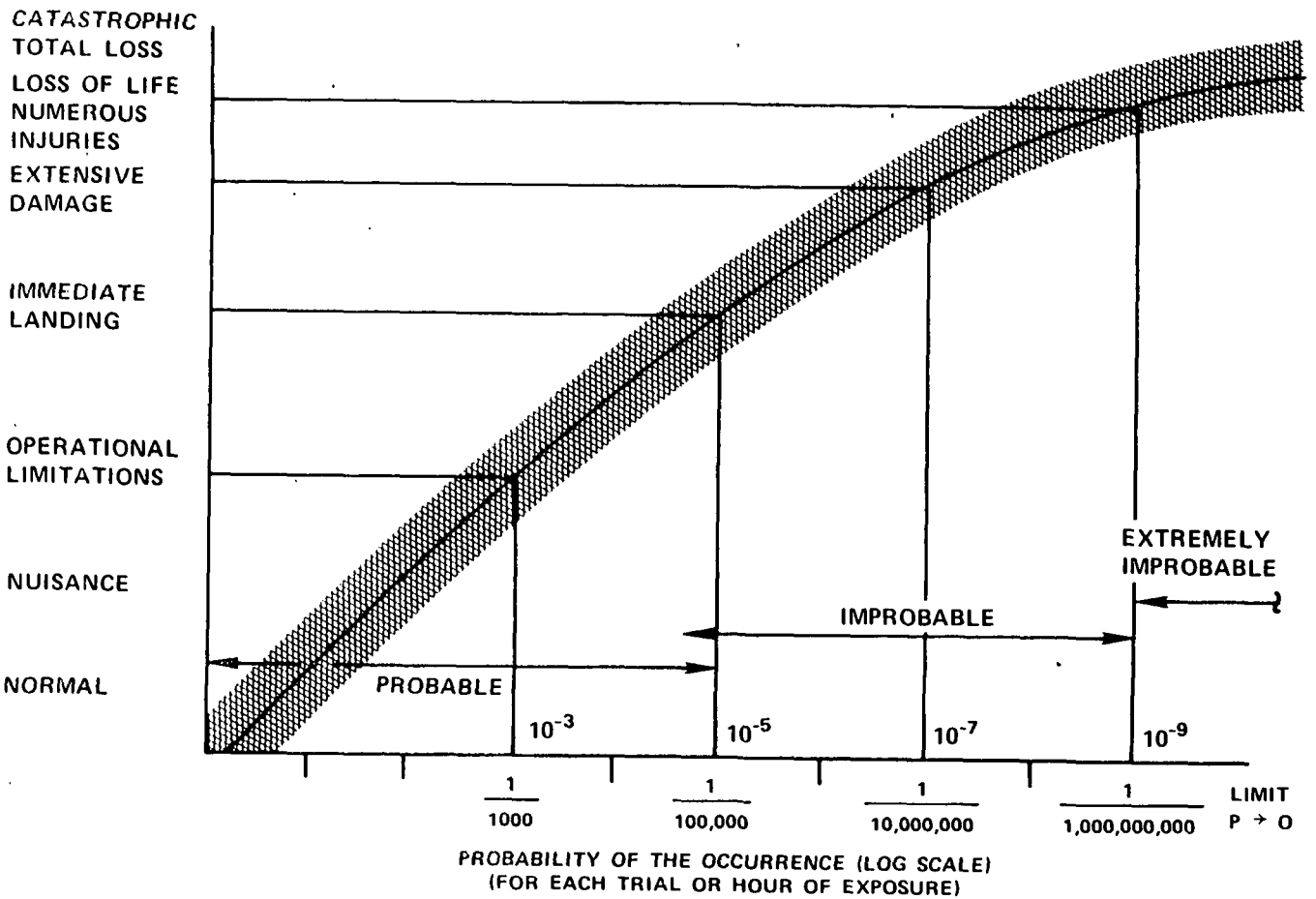
RELIABILITY REQUIREMENTS FOR TYPICAL FLIGHT CONTROL



* PILOT RATING OF UNACCEPTABLE– MARGINALLY CONTROLLABLE

** PILOT RATING OF MODERATELY OBJECTIONABLE CONTROLLABILITY

M. Perie
FAA



RELATIONSHIP BETWEEN THE CONSEQUENCE OF FAILURE
AND THE PROBABILITY OF THE OCCURRENCE

JUSTIFICATION

1. COMPLEX DFCS ARE BEING INCORPORATED ON COMMERCIAL AIRCRAFT
2. LACK OF DFCS V&V CRITERIA AND EXPERIENCE
3. PRESENT DFCS V&V METHODS ARE COSTLY, BRUTE FORCE, UNREPEATABLE AND UNQUANTIFIED
4. ADVANCED DFCS V&V TECHNIQUES HAVE NOT BEEN ADEQUATELY EVALUATED (TECHNICAL AND COST)
5. IMPORTANT FOR FUTURE AMES V/STOL AIRCRAFT REQUIRING CRITICAL DIGITAL SAS/AFCS
6. SUPPORT NATIONAL GOAL OF ASSURING TECHNICAL LEADERSHIP IN CIVIL AERONAUTICS

PRESENT V&V TECHNOLOGY – BRUTE FORCE

STRENGTHS	WEAKNESSES
• COMPREHENSIVE	• LACK OF CONSISTENCY – NOT A METHODOLOGY
• EXTENSIVE EXPERIENCE	• LACK OF QUANTITATIVE COVERAGE
	• HIGH COST

M. Perie
FAA
5 of 13

THE STATE-OF-THE-ART OF THE SOFTWARE DEVELOPMENT PROCESS

- UNIVERSITIES AND SOFTWARE RESEARCH HOUSES HAVE CONCENTRATED THEIR EFFORTS IN IMPLEMENTING ADVANCED DEVELOPMENT AND VERIFICATION TECHNIQUES.
- ONLY MINOR EFFORTS HAVE BEEN MADE TO:
  - APPLY THE ADVANCED TECHNIQUES TO PRACTICAL CASES,
  - QUANTITATIVELY ASSESS THE TECHNICAL AND FINANCIAL IMPACT OF THE TECHNIQUES ON THE DEVELOPMENT/VERIFICATION PROCESS,
  - REFINE THE USER INTERFACE.

RESULTS: THE CREATION OF THE PRESENT TECHNOLOGY GAP BETWEEN THE SOFTWARE HOUSES AND THE POTENTIAL USERS AND THE TOTAL LACK OF ENTHUSIASM WHICH AIRFRAME AND AVIONICS COMPANIES DISPLAY TOWARDS THE NEW TECHNOLOGY.

OBJECTIVES

- QUANTITATIVE ASSESSMENT OF THE TECHNICAL CAPABILITIES OF ADVANCED SOFTWARE VERIFICATION & DEVELOPMENT TECHNIQUES
- QUANTITATIVE ASSESSMENT OF THE COST OF ADVANCED SOFTWARE VERIFICATION & DEVELOPMENT TECHNIQUES
- DEVELOP AN OPTIMUM CONFIGURATION FOR THE DEVELOPMENT AND VERIFICATION OF CRITICAL DFCS SOFTWARE

FLIGHT SOFTWARE PROBLEM SURVEY

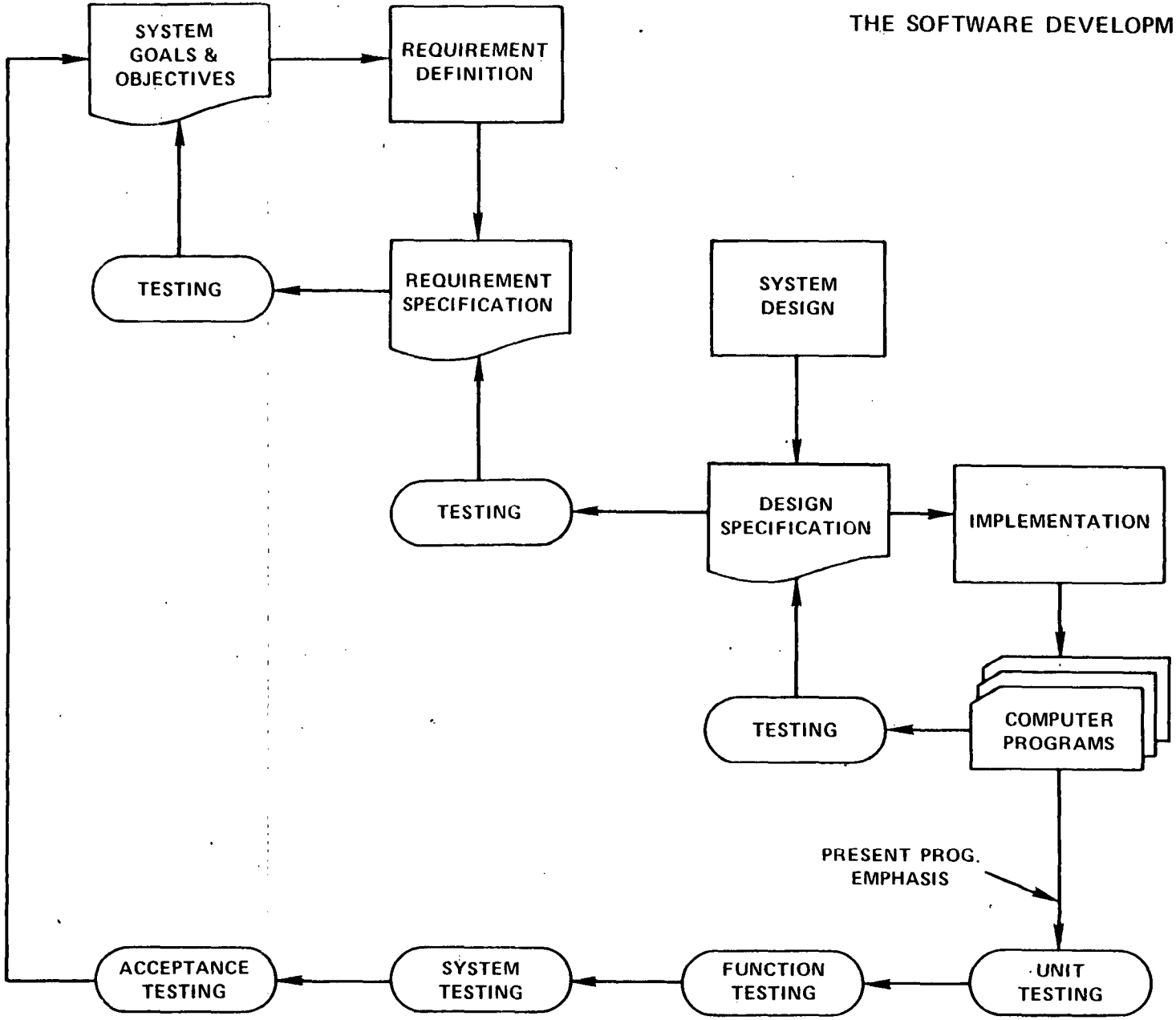
PERFORM A FLIGHT SOFTWARE DATA SURVEY, LIMITED TO DIGITAL AVIONICS SYSTEMS TO ASSESS:

- THE TYPE AND FREQUENCY OF SW ERRORS MOST LIKELY TO BE PRESENT IN DFCS
- THE COST TO DETECT, CORRECT, AND DOCUMENT THE ERRORS AND TO REVERIFY THE SYSTEM
- THE PHASE OF THE DEVELOPMENT PROCESS WHERE THE ERRORS WERE INTRODUCED
- THE SOFTWARE, HARDWARE, AND HUMAN RESOURCES REQUIRED
- THE ERRORS MOST LIKELY TO ESCAPE VERIFICATION

AUTOMATED SOFTWARE VERIFICATION TOOLS

- SOFTWARE ERRORS WILL BE SEEDED IN THE LABORATORY TEST BED, CONSISTENT WITH THE SURVEY ERROR DATA.
- SELECTED TOOLS WILL BE APPLIED TO THE SEEDED SOFTWARE TO ASSESS THEIR DETECTION CAPABILITY FOR EACH ERROR TYPE.
- THE TOOLS WILL BE ENHANCED TO INCREASE THE ACHIEVABLE COVERAGE. THE LIMITS AND CAPABILITIES OF EACH TOOL/TECHNIQUE WILL THEN BE QUANTITATIVELY ASSESSED.
- THE TOOLS WILL BE ASSEMBLED IN AN INTEGRATED PACKAGE. A METHODOLOGY WILL BE PROPOSED WHICH CONSISTS OF AN OPTIMUM COMBINATION OF ADVANCED AND CONVENTIONAL TECHNIQUES.
- THE COST TO DETECT, CORRECT, AND PREVENT ERRORS WITH THE PROPOSED METHODOLOGY WILL THEN BE QUANTITATIVELY ASSESSED AND COMPARED WITH PRESENT COSTS.

THE SOFTWARE DEVELOPMENT CYCLE

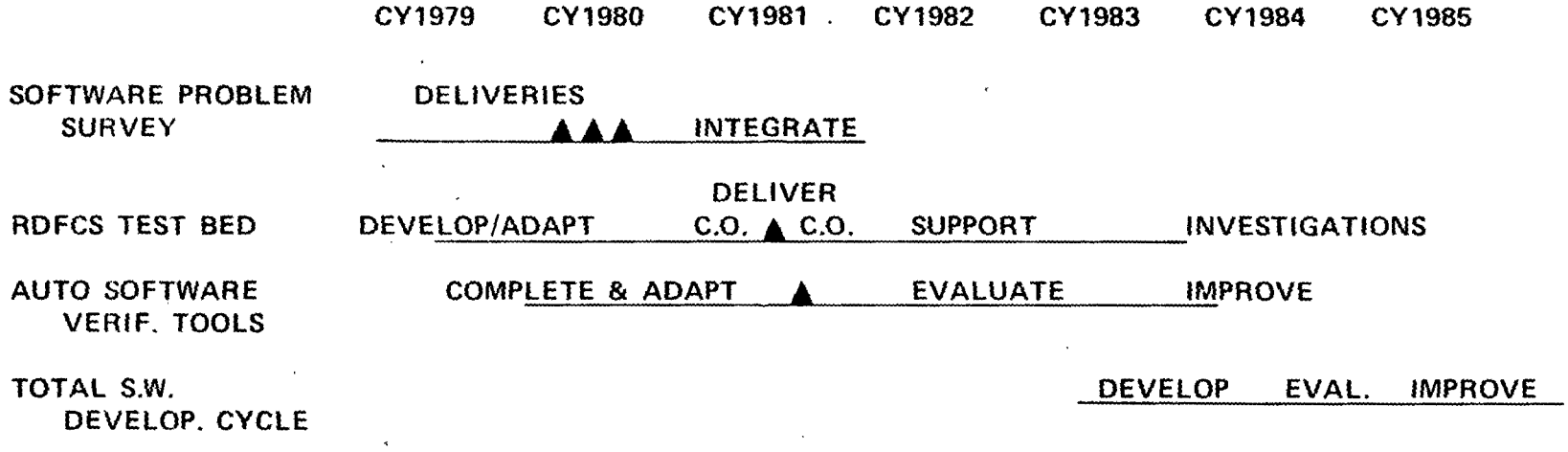


THE SIGNIFICANCE OF THE EARLY PHASES OF THE SW DEVELOPMENT CYCLE

- THE RELIABILITY OBJECTIVES OF CRITICAL FLIGHT SYSTEMS REQUIRE AN ADVANCED APPROACH TO DEFINE THE SPECIFICATIONS, THE REQUIREMENTS, AND THE DESIGN OF THE SYSTEM.
- A SIGNIFICANT PERCENTAGE OF ERRORS ARE INTRODUCED IN THE EARLY PHASES OF THE SW DEVELOPMENT PROCESS.
- THESE ERRORS ARE THE MOST DIFFICULT TO DETECT AND THE MOST COSTLY TO CORRECT.

KEY SOFTWARE DEVELOPMENT ELEMENTS

- FORMAL SPECIFICATIONS AND REQUIREMENTS LANGUAGES
- GRAPHIC TECHNIQUES FOR SYSTEM DESCRIPTIONS
- ADVANCED DEVELOPMENT ENVIRONMENT


CY1979
CY1980
CY1981
CY1982
CY1983
CY1984
CY1985
SOFTWARE PROBLEM SURVEY
DELIVERIES
INTEGRATE
RDFCS TEST BED
DEVELOP/ADAPT
DELIVER C.O.
C.O.
SUPPORT
INVESTIGATIONS
AUTO SOFTWARE VERIF. TOOLS
COMPLETE & ADAPT
EVALUATE
IMPROVE
TOTAL S.W. DEVELOP. CYCLE
DEVELOP
EVAL.
IMPROVE

DESIGNING A SOFTWARE DATA COMPENDIUM

Jon Martens
L. Duvall
IIT Research Institute
Box 1355, Branch PO
Rome, NY 13440

Many of the engineering disciplines have organized technical data into handbooks or data compendiums. These reference works are used by engineers for a large variety of engineering tasks throughout an engineered product's entire life cycle. Electronic engineers, for example, are able to use compendiums of failure data in both the design and maintenance of electronic components, circuits, and systems. Unfortunately, there is a distinct lack of such engineering aids for use by members of the software engineering community.

As part of its role as a software engineering information analysis center, the Data and Analysis Center for Software is currently designing a software engineering data compendium. Hopefully, this compendium will serve as a start in filling in some of the gaps that exist in software engineering data.

The paper will highlight some of the more critical areas that are considered during the design of an engineering reference guide such as a data compendium. These areas include:

- Identifying potential subject areas for the compendium.
- Identifying and evaluating potential data sources.
- Identifying key data elements and determining the most effective methods of data organization and presentation for use by software engineers.
- Summarizing the data at a level that minimizes volume while optimizing information value.
- Using automated tools to effectively and efficiently manage, organize, and report the data.

DESIGNING A SOFTWARE DATA COMPENDIUM

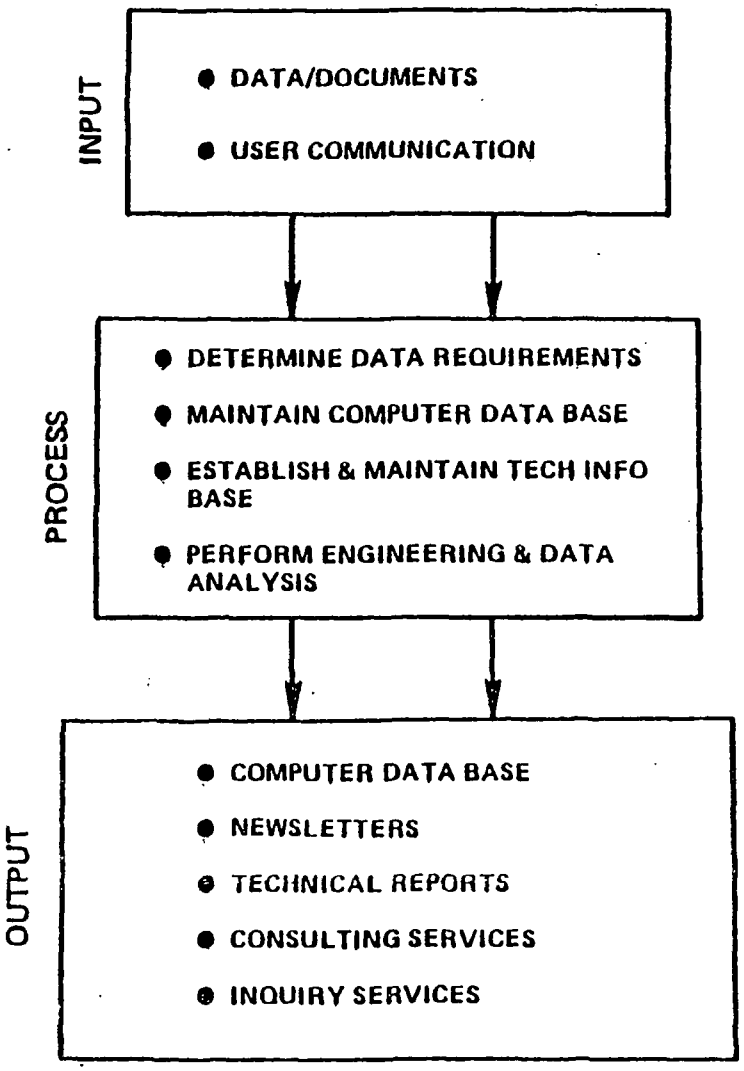
FOURTH SOFTWARE ENGINEERING WORKSHOP

NOVEMBER 19, 1979

JON MARTENS
LORRAINE DUVALL

IIT RESEARCH INSTITUTE

DATA & ANALYSIS CENTER FOR SOFTWARE



IITRI

L. Duvall
IITRI

DESIGNING A S/W DATA COMPENDIUM

- ENGINEERING DISCIPLINES USUALLY HAVE HANDBOOKS
- EXAMPLE:
  ELECTRONIC ENGINEERS - COMPONENT FAILURE HANDBOOKS
- SOFTWARE ENGINEERS DO NOT HAVE HANDBOOKS OR
  DATA COMPENDIUMS
- AS AN IAC, DACS IS DESIGNING A SOFTWARE ENGINEERING
  DATA COMPENDIUM
- GOAL:

  TO SERVE AS A START FOR A SOFTWARE ENGINEERING
  HANDBOOK

L. Duvall
IITRI

CRITICAL AREAS IN DATA COMPENDIUM DESIGN

- IDENTIFYING SUBJECT AREAS
- IDENTIFYING AND EVALUATING DATA SOURCES
- IDENTIFYING KEY DATA ELEMENTS AND METHODS OF ORGANIZATION
- SUMMARIZING THE DATA AT AN OPTIMUM LEVEL
- USING AUTOMATED TOOLS

IDENTIFYING POTENTIAL SUBJECT AREAS

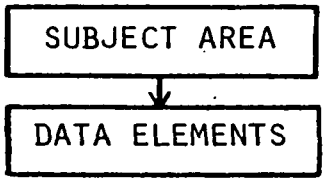
- POSSIBLE SUBJECT AREAS
  - FAILURE/TEST DATA
  - COST/PRODUCTIVITY DATA
  - COMPONENT DATA
  - PROJECT/MANAGEMENT DATA

IDENTIFYING & EVALUATING POTENTIAL DATA SOURCES

- POTENTIAL SOURCES
  - BSDS (6SPR/SMN RELATED DATASETS)
  - RADC PRODUCTIVITY DATABASE (PRODUCTIVITY)
  - NASA SEL DATABASE (PRODUCTIVITY, COMPONENT DATA, PROJECT DATA, SPR/SMN, TEST RESULTS)
  - FAA/CSC DATABASE (PRODUCTIVITY, SPR/SMN)
- EVALUATION CRITERIA
  - AVAILABILITY
  - COMPLETENESS
  - APPLICABILITY
  - CONSISTENCY
  - MEDIA (MACHINE READABLE)

IDENTIFYING KEY DATA ELEMENTS & METHODS OF ORGANIZATION

- KEY DATA ELEMENTS DERIVED FROM SUBJECT AREAS



- ORGANIZATION DEPENDENT ON SUBJECT & SUMMARIZATION LEVELS

SUMMARIZING DATA AT OPTIMUM LEVEL

- DATABASE "DUMPS" RESULT IN INFORMATION OVERLOAD
- HIGH SUMMARIZATION LEVELS RESULT IN INFORMATION LOSS
- BALANCE BETWEEN INFORMATION OVERLOAD & LOSS MUST BE BALANCED

USING AUTOMATED TOOLS

- DATABASE MANAGEMENT SYSTEMS - MDQS
- STATISTICAL PACKAGES - SPSS
- TEXT PROCESSING TOOLS - RUNOFF

AVERAGE PROGRAMMER HOURS

COMPONENT TYPE	DESIGN			DEVELOPMENT			TESTING		
	CREATE	READ	REVIEW	CODE	READ	REVIEW	MOD	INTEQ	REVIEW
String Processing									
Scientific Command & Control									
Business & Financial									
Database Application									
.									
.									
.									
.									
.									
.									

SOFTWARE V & V TOOLS – AN ASSESSMENT PROGRAM

Dr. Pio V. de Feo
NASA Ames Research Center
Moffett Field, CA 94035

ABSTRACT

Significant progress has been made in recent years in the area of software verification tools. However, their utilization in the verification of Digital Avionics Systems has been very limited. The primary reason for this is the lack of data proving the cost effectiveness of these tools.

NASA has started a program to fully and quantitatively assess the impact of including advanced software verification and validation (V & V) tools in the verification of Digital Flight Control Systems (DFCS) software for commercial applications; the technical capabilities as well as the financial implications of these tools will be analyzed. An outline of the program and of its primary motivators is presented.

BACKGROUND

A significant shift from analog to Digital Flight Controls is occurring in recent civil aircraft developments due to improvements in life cycle cost and the ability to perform complex functions. These systems have great potential for improving aircraft performance and cost of operation; however, the reliability requirements can be very high for systems which perform critical functions such as low visibility landings, and relaxed static stability, etc. The reliability requirements are established by the ceritification agencies based upon:

1. The probability that the loss of the system will induce a catastrophe (loss of life);
2. The exposure time of the system.

As an example, the reliability required for critical autoland systems, which have an exposure time of less than one (1) minute per flight, is significantly less than the reliability requirements for flight critical systems, like advanced relaxed static stability, which is 10^{-9} failures/hour. These reliability requirements are satisfied by configuring the hardware in redundant configurations; however, although not explicitly stated, the software is assumed to be free of errors and is handled as a component with a zero failure rate; as a result, the software in most redundant configurations is the primary source of single point failures. The verification of the software of flight critical DFCS is therefore an extremely challenging task; the challenges are further increased by the need to keep the verification cost within reasonable limits.

The present V & V technology for DFCS is primarily based upon extensively exercising the systems in closed-loop, real-time simulations where the actual operational environment is simulated to a degree of fidelity which reflects the objectives of the test. The technique is very comprehensive; in fact, it is capable of detecting any type of errors, from specification errors to coding errors. The avionics and airframe companies have significant experience in the use of this technique and practically every digital system which has been flown was verified primarily using this technique; however, the technique is not perfect and errors which were undetected during the verification were

later found during the operation of those systems. The technique cannot guarantee consistent results; and, actually, the quality of the test is in large part dependent on the intelligence and dedication of the analysts who performed the tests. The actual test coverage is not quantifiable; and, in fact, much time can be spent in testing over and over again the same programs, routines, or logic paths while others have never been executed. Finally, the technique is expensive; sophisticated iron-bird simulations are costly to develop and operate and their use should be limited to very specific and well planned test objectives.

PROGRAM OBJECTIVES

In recent years significant progress has been made towards a better understanding of the entire software development process and the challenges involved in each phase of this process. It is generally agreed that the early phases of the process, the specification, the requirement, and the design phase have the greatest impact on the quality and the correctness of the final product. However, the most significant progress has been made in the area of the software verification tools which can be applied only after code generation; these tools are generally classified as static tools, which do not require the flight software to be executed, and dynamic tools which do require it. The theoretical feasibility of many software verification tools has been demonstrated, several have already been developed, and some have also been applied to software programs of medium size. In spite of this obvious progress, the airframe and avionics companies show very little enthusiasm for the new technology and are reluctant to include it in the verification process. The primary reasons for this are: The tools are poorly understood; very limited quantitative data are presently available relative to their error detection capabilities and their operating cost; the poor level of development of most tools; the poorly designed operational environment.

The primary purpose of this program is to perform a quantitative assessment of the operating cost and of the technical capabilities of these tools within the context of Digital Flight Control Systems.

THE PROGRAM PLAN

To meet the program objectives, an analysis must be performed to clearly understand the technical capabilities and limitations and the cost of the present verification technology. The same analysis must then be performed relative to the advanced software verification tools. At the end, the relative advantages and disadvantages of each technique will be defined and an integrated methodology, which includes conventional and advanced techniques, will be proposed.

The program is structured in three phases:

Phase 1: During this phase a quantitative assessment of the present development and verification technology of Digital Avionics Systems will be made. Data will be gathered exclusively from the analysis of recent development programs of Digital Avionics Systems. Specifically, the following will be determined:

P. deFeo
NASA/AMES
2 of 4

(a) The type and frequency of software errors most likely to be present;

(b) The cost to detect, correct, and document these errors and the procedures and techniques used;

(c) The errors most likely to escape detection.

These data will provide a meaningful benchmark, representative of the present technology, against which the capabilities and cost effectiveness of the advanced techniques will be rated

Phase 2: During this phase, a quantitative assessment of the error detection capabilities of the software verification tools will be made. The following is an outline of the activities required to accomplish this objective:

(a) A Digital Avionics System, representative of the near term technology for critical commercial applications, has been procured. The system will be used as a test bed for the software tools.

(b) An initial integrated set of software V & V tools, compatible with the test bed system, is currently under procurement.

(c) The flight software of the test bed system will be randomly seeded with errors consistent, in type and frequency, with the results of the analysis performed during Phase 1.

(d) The error detection capabilities of each tool will then be determined by applying the tools to the seeded software; the percentage of detected errors will be a quantitative measure of the test coverage achievable by each tool. Enhancements will be made, whenever feasible, to increase the original test coverage of each tool.

At the end of this phase, the technical limits and capabilities of the V & V tools will be quantitatively assessed. A technical recommendation for the inclusion of selected tools in the verification process of DFCS can also be made based upon:

(1) The types of errors which each tool is capable of detecting and how thoroughly and consistently each tool performs.

(2) The level of complementarity of coverage and synergism with the conventional verification techniques.

Phase 3: A quantitative assessment of the error detection capabilities of advanced software V & V tools will be performed during Phase 2. The technical capabilities must be the prime consideration for the inclusion of selected tools in the verification process of critical DFCS. However, the willingness or reluctance of the avionics and airframe companies to actually include advanced software V & V tools in their verification programs will be strongly influenced by the economical impact of the tools to the already high cost of verification. The activities in this Phase of the program will aim at a quantitative assessment of the economics of operating the tools. Major cost factors which will be analyzed are: The initial cost of procuring the tools, the cost of adapting existing tools to new environments, the number of systems over which these costs can be amortized, the cost of operating the tools, etc.

P. deFeo
NASA/AMES
3 of 4

The use of the verification tools will result in software programs which have fewer errors at the start of the system testing phase. This could appreciably decrease the efforts needed in the area of closed-loop, iron-bird simulation analysis. These simulations are very expensive to build and even more expensive to run due to the high personnel support they need. A more efficient use of these facilities could be a major cost savings factor induced by the utilization of the tools. Additional savings should be realizable because the tools promote error detection at a very early phase of the coding process; this minimizes the economical impact of the errors and their documentation process. If the errors were detected later, the process would be significantly more expensive because the configuration would be more formally controlled.

All these economic factors will be quantified at the end of this phase.

CONCLUSION

The objective of this program is to fully and quantitatively assess the impact of software V & V tools to the verification of Digital Flight Control Systems for critical applications. The intrinsic complementarity of the tools and their synergism with conventional verification techniques should make feasible and attractive the development of an integrated verification package so that high quality software can be generated at a reasonable cost. An effort will be made, within the scope of this program to specify that package.

P. deFeo
NASA/AMES
4 of 4

PANEL #3

EXPERIMENTS IN METHODOLOGY EVALUATION

P. Hsia, University of Alabama
S. Sheppard, GE
W. Fujii, DDI

P. Hsia
University of Alabama

Experiments in evaluating a step-by-step software development process have been conducted with University students at the University of Alabama. Results and indicators of these experiments will be discussed.

STEP	PRODUCT
1. Problem Analysis	• Input • Output • Relationships • Sample Computation
2. Solution Design	• Flowchart • Data Flow
3. Test Planning	• Test Case Summary • Input • Predicted Output
4. Peer Review	• Sign Off After Approval
5. Coding	
(a) Translate Into Programming Language	• Code
(b) Keypunch/Terminal Entry	• Cards/File
(c) Compilation	• Syntax-Error Free Code
6. Testing	• Test Cases' Execution Results
7. Acceptance	
• Acceptance Testing	• Instructor's Test Cases
• Notebook Consolidation	• Merge of All Products

Figure 1. Disciplined Framework

P. Hsia
Univ. of Ala.

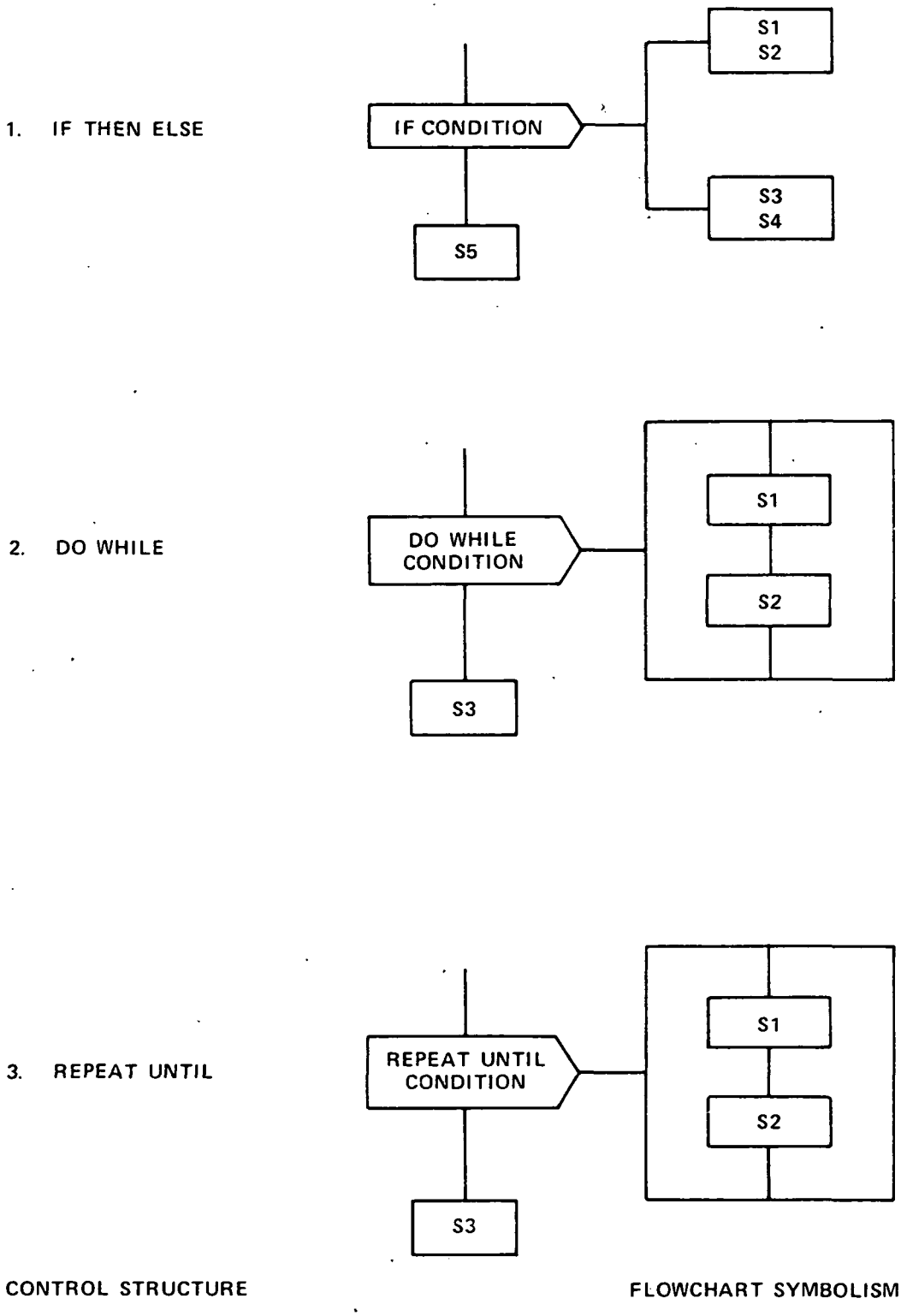


Figure 2. HOS Flowchart Symbolism

P. Hsia
Univ. of Ala.

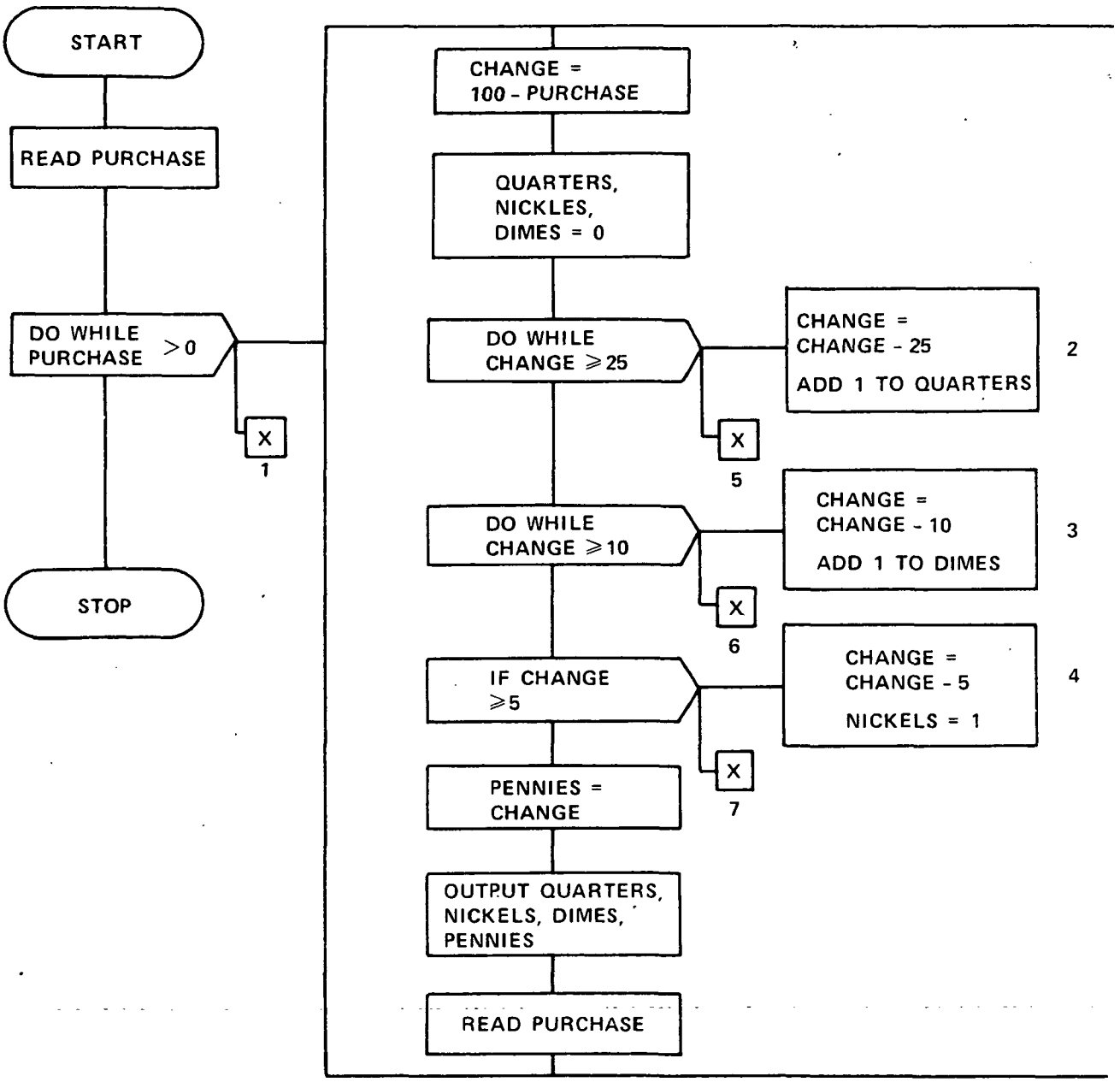


Figure 3(a). Sample Flowchart

P. Hsia
Univ. of Ala.

Test Case	Input	Branch							Output				
		1	2	3	4	5	6	7	Q	D	N	P	
1	0	X							–	–	–	–	*
2	55		X	X				X	1	2	0	0	
	92				X	X	X		0	0	1	3	
	0	X							–	–	–	–	*

*No Output in Branch 1

Figure 3(b). Test Case Matrix for Labeled Flowchart of Figure 3

P. Hsia
Univ. of Ala.

```
            READ (5,-)IPUR

10          CONTINUE
            IF(.NOT.(IPUR.GE.0))GO TO 11
                ICHNG = 100-IPUR
                IQUAR = 0
                IDIME = 0
                INICK = 0

20              IF(.NOT.(ICHNG.GE.25))GO TO 21
                    ICHNG = ICHNG-25
                    IQUAR = IQUAR+1
                GO TO 20

21              CONTINUE
30              IF(.NOT.(ICHNG.GE.10))GO TO 31
                    ICHNG = ICHNG-10
                    IDIME = IDIME+1
                GO TO 30

31              CONTINUE
                IF(.NOT.(ICHNG.GE.5))GO TO 41
                    ICHNG = ICHNG-5
                    INICK = 1

41              CONTINUE
                IPEN = ICHNG
                WRITE(6,-)IQUAR,IDIME,INICK,IPEN

                READ(5,-)PUR


            GO TO 10
11          CONTINUE
```

Figure 4. Structured FORTRAN Code for Flowchart of Figure 3.

P. Hsia
Univ. of Ala.

RUNLOG

For Name:_____

Run Number
Date
Time
Cost
Purpose

I. Analysis

II. Corrective Actions

III. Lessons Learned

Figure 5(a). Run Log Form

P. Hsia
Univ. of Ala.

TIME LOG

FOR

Project:_____ Name:_____

Step \ Time	Actual Effort (in Man-Hours)	Actual Date of Completion
Problem Analysis Analysis		
Solution Design		
Test Planning		
Peer Review		
Coding: • Translation Into Programming Language • Keypunching • Compilation		
Testing		
Acceptance: • Acceptance Test • Notebook Consolidation		

Figure 5(b). Time Log Form

STEP	PRODUCT
1. Flowcharting	• Flowchart
2. Coding	
(a) Translate Into Programming Language	• Code
(b) Keypunch/Terminal Entry	• Cards/File
3. Testing and Debugging	• Complete Program with Test Cases and Results
4. Documentation	• Flowchart • Complete Programs • Test Cases and Results

Figure 6. Conventional Programming

Term	Assignment	Average Times in Hours		Ratio of Times
		Disciplined Approach	Conventional Approach	
1	1	8.9	7.6	1.17
	2	8.4	7.3	1.15
	3	23.4	20.2	1.16
2	1	10.3	8.1	1.27
	2	14.1	13.5	1.05
	3	23.9	20.1	1.19

Figure 7. Comparison of Times in Two Approaches

P. Hsia
Univ. of Ala.

		Conventional Approach			Disciplined Approach		
Term	Programs	Total Programs	Programs With One or More Errors	% of Error-Free Programs*	Total Programs	Programs With One or More Errors	% of Error-Free Programs*
Term 1	Prog 1	36	5	86%	20	0	100%
	Prog 2	34	6	82%	20	0	100%
	Prog 3	9	6	33%	14	4	71%
	Prog 4	20	16	20%	16	9	43%
	Overall Total	99	33	67%	70	13	81%
Term 2	Prog 1	20	7	65%	68	7	90%
	Prog 2	22	5	77%	59	3	95%
	Prog 3	16	8	50%	31	13	58%
	Overall Total	58	20	65%	158	23	85%

*Should read % of error-free programs with respect to the selected sets of comprehensive test cases.

Figure 8. Comparative Data for Logic Errors

EXPERIMENTAL RESULTS ON SOFTWARE DEBUGGING

Sylvia B. Sheppard
Phil Milliman
Bill Curtis
Software Management Research
Information Systems Programs
General Electric Company
1755 Jefferson Davis Highway
Arlington, VA 22202

INTRODUCTION

Debugging programs is one of the most expensive, time-consuming activities in the development of a software system. Only a few laboratory experiments have investigated the relative difficulty of locating different types of bugs or the most effective search strategies. Youngs (1974) found that experience contributed to differences among types of errors made in a construction experiment. Wescourt and Hemphill (1978) described a model of the debugging process, but the model was not entirely supported by the available data. Gould and his associates (Gould and Drongowski, 1974; Gould, 1975) found that the type of bug influenced debugging performance on short programs. Specifically, assignment bugs were more difficult to locate than array or iteration bugs, probably because the former required a greater understanding of the algorithm used by the prograr

The difficulty of debugging a program may be associated with coding practices used during its development. One factor which may influence the ease of finding a bug is the complexity of a program's control flow. Two previous experiments by the authors investigated the effects of structured control flow in understanding and modification tasks (Sheppard, Curtis, Borst, Milliman, and Love, 1979). Programmers performed their tasks more efficiently on code which exhibited a straightforward, top-down control flow than on an unstructured, convoluted control flow. A rigorously structured control flow (Dijkstra, 1972) did not produce significantly better performance than a naturally structured version which allowed limited unstructured constructs (e.g., exits from loops). Thus the overall top-down quality of the control flow appears to influence performance, while minor deviations from the tenets of structured code do not appear to influence performance significantly. This result may reflect the innate awkwardness of implementing strictly structured code in standard Fortran.

Factors other than the structuredness of the control flow may influence the complexity of a computer program and, thus, the difficulty programmers experience in performing their tasks. Some of these factors have been quantified in the software complexity metrics developed by Halstead (1977) and McCabe (1976). Halstead's metric purportedly represents the number of mental discriminations involved in developing a program, while McCabe's metric measures the number of elementary control path segments comprising a program. In experiments on understanding and modification, these software complexity metrics were evaluated for their usefulness as predictors of programmer performance (Curtis, Sheppard, Milliman, Borst, and Love, 1979). The results observed in those experiments were modest. The correlations in the raw data were not large, and the number of lines of code usually predicted programmer performance better than the Halstead or McCabe metrics. Several limitations in the experimental procedures employed to obtain the data may have produced these results. First all of the programs studied were short

(35–55 lines of code). The limited range of metric values calculated on programs of this length may not have been sufficient for an adequate test of the predictive worth of the metrics. Second, individual differences among programmers exerted significant effects on the results obtained. When the data from the first experiment were transformed in an attempt to control for differences among programs and programmers, a correlation of -0.73 ($p < 0.001$) was obtained between the performance criterion and Halstead's E. However, the issue is not whether theories can be validated with mystical transformations of data, but whether the results of these heuristic transformations can be replicated in an experiment designed to overcome the limitations of previous research.

The present experiment evaluated the difficulty of locating three types of errors under controlled programming conditions. In order to compare the effects on performance of different methods of structuring code, programs in the present experiment were implemented in three types of control flow, all of which exhibited a generally top–down flow. This experiment also evaluated the ability of software complexity metrics to predict performance over a wider range of program sizes. To investigate the effects of length, the three programs in this experiment were subdivided into functional subroutines so that they could be presented in three different lengths: approximately 50, 125, and 200 lines of code. Finally, the present experiment attempted to relate programming performance to experiential factors, such as familiarity with other programming languages or relevant programming tools and concepts.

METHOD

Participants

Fifty–four professional programmers at six different locations participated in this experiment. Thirty were civilian employees, while 24 were employees of the military. The participants averaged 6.6 years of professional experience programming in Fortran, ranging from 1/2 year to 25 years (SD = 6.1).

Experimental Design

In order to control for individual differences in performance, a within–subjects, 3^4 factorial design was employed. Three types of control flow were defined for each of three programs, and each of these nine versions was presented in three lengths with three different bugs, for a total of 81 different experimental conditions. The first 27 participants each saw three of the programs, exhausting the 81 conditions (Fig. 1). The second set of 27 participants replicated the conditions exactly except that the order of presentation of the tasks was different in each case.

Learning effects were expected on the basis of results obtained in previous experiments of this type (Sheppard, Curtis, Borst, Milliman, and Love, 1979; Sheppard and Love, 1977). Therefore, the order of presentation of conditions was counterbalanced to assure that each level of each independent variable appeared as the first, second, or third task an equal number of times.

Procedure

A packet of materials prepared for each participant included: (1) written instructions on the experimental tasks, (2) a short tutorial of commands used in Fortran 77, (3) a short preliminary task (Appendix A), (4) three experimental tasks, and (5) a questionnaire concerning previous experience.

PROGRAM	LENGTH	NATURALLY STRUCTURED			GRAPH-STRUCTURED			FORTRAN 77			← CONTROL FLOW
		1	2	3	1	2	3	1	2	3	← BUG
1 ROOTS	SHORT	1	23	12	20	15	3	18	2	26	
	MEDIUM	19	11	7	14	9	25	8	22	17	
	LONG	10	4	27	6	24	15	21	16	5	
2 ACCT	SHORT	13	8	21	7	27	16	24	10	9	
	MEDIUM	5	26	15	23	18	4	12	6	20	
	LONG	22	14	2	17	1	19	3	25	11	
3 GRADER	SHORT	25	17	6	11	5	22	4	19	14	
	MEDIUM	16	3	24	2	21	10	27	13	1	
	LONG	9	20	18	26	12	8	15	7	23	

EACH CELL REPRESENTS ONE OF THE THREE TASKS GIVEN TO A PARTICIPANT

Figure 1. Assignments of 27 Participants in One Replication of the Experimental Design

All tasks included input files, a listing of the Fortran program with the embedded bug, a correct output, and the erroneous output produced by this program. All differences between the correct and erroneous output were circled on the erroneous output. Also included were explanatory descriptions of any subroutines or functions not presented in the listing but referenced by the program.

The 54 participants were divided into two groups of 27, each of which represented a complete replication of the design. Within a group all participants were given the same preliminary task. Group 1 worked with an algorithm to find the greatest common divisor of two numbers and Group 2 was given a simple sort algorithm. These preliminary tasks were provided to reduce learning effects on the experimental tasks and to provide a basis for comparing the abilities of the participants to perform a task of this nature.

Following the initial exercises, participants were presented with three separate programs comprising their experimental tasks. Participants were allowed to work at their own pace, signalling the experimenter when they believed they had identified and corrected the bug. The experimenter verified all corrections, and in the case of a mistake the participant was instructed to try again until the task was successfully completed. The maximum time participants were allowed to work on a particular program was 45 minutes for the preliminary task and 60 minutes for each experimental task. Time was measured to the nearest minute.

Independent Variables

Program. Three programs were selected for the generality of their content and their understandability to programmers. The first program sorted and categorized alphabetic response data to a questionnaire (Veldman, 1967). The second program, an accounting routine, produced income and balance statements (Nolen, 1971). Program 3 kept track of students' test grades and calculated their semester averages (Brooks, 1978). All programs were tested prior to the experiment.

Length. The inclusion of additional subroutines made it possible to present each program in three different lengths. The shorter programs had 25–75 statements, medium programs contained 100–150 statements, and the longer programs contained approximately 175–225 statements. (One Fortran 77 version exceeded the 225 line limit by 8 lines because of the number of ELSE and ENDIF statements required.)

Program listings included a two or three line explanation of any routine or function that was called by a program but not presented in the experimental materials. Participants were told to assume that missing routines worked correctly. All of the input and output files were presented regardless of the length of the program. That is, for the shorter version, some of the input was read in and some of the output was produced by subroutines which were not presented.

Complexity of Control Flow. Three versions of control flow performing identical tasks were defined for each program. Two types of structures were implemented in Fortran IV, naturally structured and graph-structured. A third version was written in Fortran 77 (Brainerd, 1978), which includes the IF-THEN-ELSE, DO-WHILE, and DO-UNTIL constructs.

The Fortran 77 version of each program was implemented in a precisely structured manner. All flow proceeded from top to bottom, and only three basis control constructs were allowed: the linear sequence, structured selection, and structured iteration (Fig. 2).

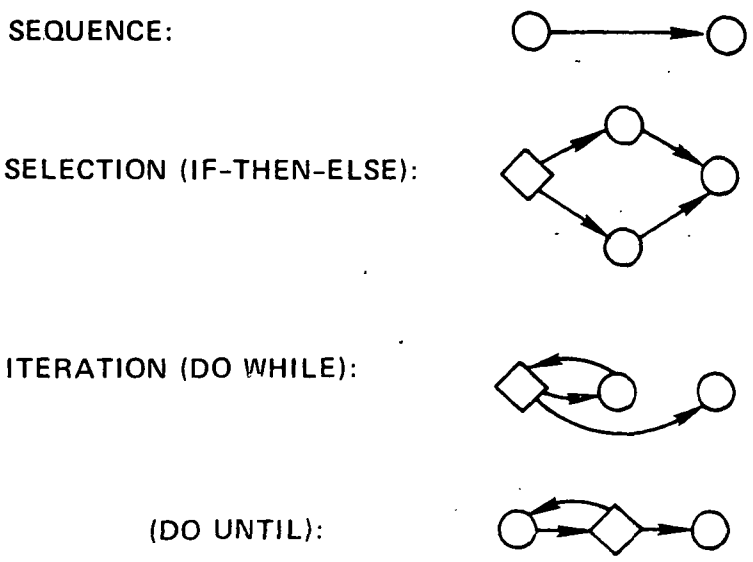


Figure 2. The Basic Structured Constructs

The graph-structured version of each program was implemented in Fortran IV from the Fortran 77 version, replacing the special constructs but producing code for which the control flow graphs of the two versions were identical. All nested relationships could be reduced through structured decomposition to a linear sequence of unit complexity. A full discussion of reducibility is presented by McCabe (1976).

Structured constructs are awkward to implement in Fortran IV (Tenny, 1974). In order to test a more naturally structured flow, limited deviations were allowed in a third version of each program. These deviations included such practices as branching into or out of a loop or decision and multiple returns. Control flow graphs and the code for a section of a routine implemented in all three versions of control flow are presented in Figures 3 and 4.

Each program was indented following the nesting patterns presented in the code. Thus, all DO loops and branching instructions were indented. For naturally structured versions, decisions were made arbitrarily about the importance of various constructions, and indenting was necessarily less standardized than for the graph-structured and Fortran 77 versions.

Type of Bug. Three types of semantic bugs were chosen from a classification developed by Hecht, Sturm, and Trattner (1978): computational, logical, and data errors. Bugs in each category were defined for each of the three programs in order to maximize the similiarity of bugs from a single category across programs. Computational bugs involved a sign change in an arithmetic expression. Logic bugs were implemented by using the wrong logical operator in an IF condition. Data bugs involved wrong index values for variables.

Each bug in this experiment was purposely designed to affect only a limited area of code. That is, each calculation containing a bug occurred near the corresponding WRITE and FORMAT statements. In no case did a bug produce errors in routines other than the one in which it was embedded, and each bug appeared in only one line of code.

Individual Differences Measures

Scores on the preliminary exercise were used as a measure of programming ability related to the experimental task. Participants were also asked to complete a questionnaire about their programming experience. The information required included specific types of experience, number of years programming professionally in Fortran, number of statements in the longest Fortran and non-Fortran programs written, the first programming language learned, and number of languages learned. In addition, various programming concepts that appeared relevant to the experimental programs were listed, and participants were asked to mark those with which they were familiar.

Complexity Metrics

Halstead's E. Using a program based on Ottenstein (1976), Halstead's effort metric (E) was computed from the source code listings of the 27 experimental programs, representing three distinct programs at three levels of structure and three different lengths. The computational formula was:

$$E = \frac{\eta_1 N_2 (N_1 + N_2) \log_2 (\eta_1 + \eta_2)}{2\eta_2}$$

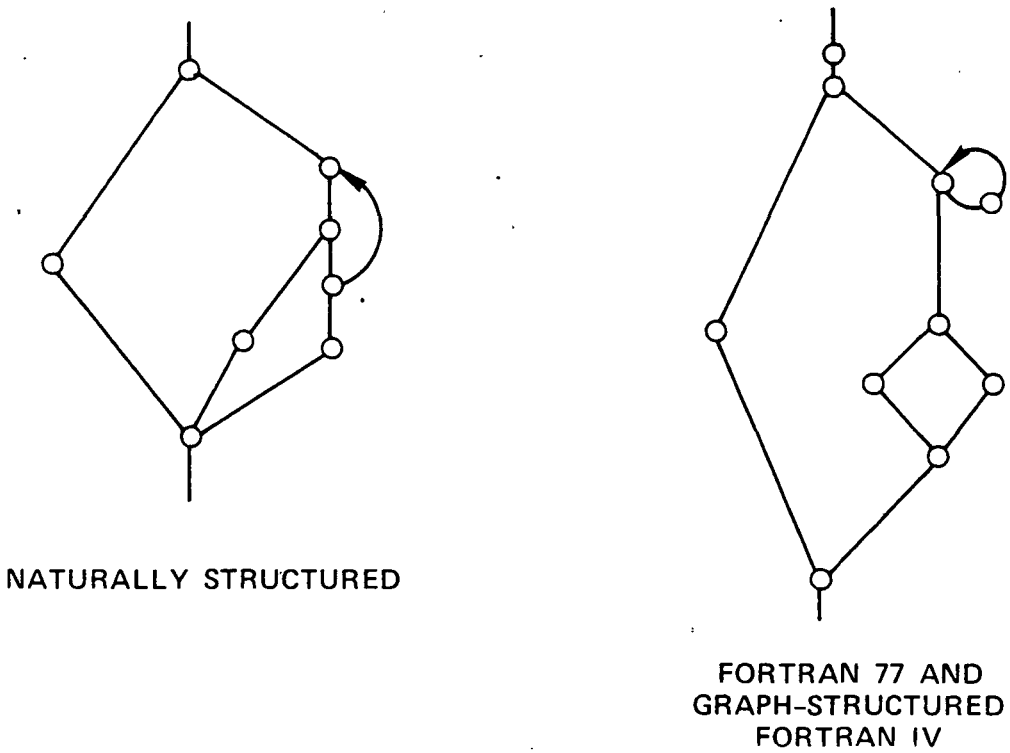


Figure 3. Control Graphs for All Versions of Control Flow

NATURALLY STRUCTURED

```
      IF (ASNUM .LT. 1 .OR. ASNUM .GT. NASSGN) GO TO 420
         DO 400 K=1,NSTUDN
            IF (CURID .EQ. ID(K)) GO TO 440
400         CONTINUE
         PRINT 410,CURID
410      FORMAT (1H0,30X,' ID NUMBER NOT IN FILE:  ',I8)
         GO TO 450
420      PRINT 430, CURID,ASNUM
430      FORMAT (1H0,30X,'ID ',I8,'ILLEGAL ASSIGNMENT',I3)
         GO TO 450
440   SCORE(K,ASNUM)=VAL
450   CONTINUE
```

GRAPH-STRUCTURED

```
      K=1
      IF (ASNUM .LT. 1 .OR. ASNUM .GT. NASSGN) GO TO 420
400      IF (CURID .EQ. ID(K) .OR. K .GT. NSTUDN) GO TO 405
            K=K+1
            GO TO 400
405      IF (K .LE. NSTUDN) GO TO 415
            PRINT 410,CURID
410         FORMAT (1H0,30X,' ID NUMBER NOT IN FILE:  ',I8)
            GO TO 450
415         SCORE(K,ASNUM)=VAL
            GO TO 450
420      PRINT 430, CURID,ASNUM
430      FORMAT (1H0,30X,'ID ',I8,' ILLEGAL ASSIGNMENT ',I3)
450      CONTINUE
```

FORTRAN 77

```
      K=1
      IF (ASNUM .GE. 1 .AND. ASNUM .LE. NASSGN) THEN
            DO 400 WHILE (CURID .NE. ID(K) .AND. K .LE. NSTUDN)
400            K=K+1
               IF (K .GT. NSTUDN) THEN
                  PRINT 410,CURID
410               FORMAT (1H0,30X,' ID NUMBER NOT IN FILE:  ',I8)
               ELSE
                  SCORE(K,ASNUM)=VAL
               ENDIF
         ELSE
            PRINT 430, CURID,ASNUM
430         FORMAT (1H0,30X,'ID ',I8,' ILLEGAL ASSIGNMENT ',I3)
         ENDIF
450   CONTINUE
```

Figure 4. Examples of the Three Types of Control Flow

where,

η_1 = number of unique operators

η_2 = number of unique operands

N_1 = total frequency of operators

N_2 = total frequency of operands

McCabe's v(G). McCabe's metric is the classical graph-theory cyclomatic number defined as:

v(G) = # edges - # nodes + 2 (# connected components). McCabe presents two simpler methods of calculating v(G): the number of predicate nodes plus 1 or the number of regions computed from a planar graph of the control flow.

Length. The length of the program was the total number of Fortran statements, excluding comments. The total number of executable statements was found to be highly correlated with number of statements ($r = 0.99$, $p \leq 0.001$).

Dependent Variable

The dependent variable was the number of minutes necessary for the participant to locate and correct the bug.

Analysis

The analysis of data was conducted in two phases. The first phase was an experimental test of the independent variables, while the second phase evaluated the software complexity metrics. In the first phase, experimental data were analyzed in a hierarchical regression analysis. In this analysis, domains of variables were entered sequentially into a multiple regression equation to determine if each successive domain significantly improved the predictive capability of the equation developed from domains already entered. Thus, the order in which domains were entered into the analysis was important. Variables representing the different conditions of experimentally manipulated variables were effect-coded (Kerlinger and Pedhazur, 1973).

The second phase of analysis investigated relationships between the time to find the bug and the metrics, Halstead's E, McCabe's v(G), and number of statements in the program. All correlations are Pearson product-moment correlations.

RESULTS

Preliminary Tasks

Group 1 (Participants 1-27) and Group 2 (Participants 28-54) were given different preliminary tasks. The two algorithms were of varying difficulty, producing significant differences in both time to completion and percent of completions. Finding the bug in the greatest common divisor algorithm required an average of 23.8 minutes with 22% failing to find the bug in 45 minutes, while the sorting algorithm required only 14.6 minutes with only 4% failing to find the bug. However, no significant differences in performance between the two groups occurred on the experimental programs.

Experimental Manipulations

The average time to locate bugs across all experimental conditions was 20.1 minutes (SD = 16.2). All but six of the 162 experimental tasks comprising this experiment were completed successfully during the allotted 60 minutes. These six conditions were not associated with any particular factor.

Despite the use of a preliminary task to familiarize the participants with the experiment, a significant order effect occurred ($p \leq 0.04$), indicating that learning took place during the first of the three experimental tasks (Fig. 5).

Results of a hierarchical regression analysis of the independent variables on the time to find the bug are presented in Table 1. Differences in solution time for the three programs were significant ($p \leq 0.01$). Finding the bug in the accounting program required an average of 15.1 minutes, 20.0 minutes in the program that sorted questionnaire data, and 25.0 minutes in the grade-scoring program. Increasing the length of the programs had a modest effect ($p \leq 0.06$) on the time to locate and correct the error. The average time for the short program was 16 minutes, while the medium and long programs required a mean of 21 and 23 minutes, respectively.

Averages for the three error categories were not significantly different from one another. However, a very large interaction occurred between type of bug and program (Fig. 6). This interaction accounted for the largest percent of variance (26%) of any of the experimental relationships studied. No significant differences in performance resulted from the three types of control flow.

Software Complexity Metrics

Intercorrelations among the three measures of software complexity were computed from the 27 different versions of the programs at both the subroutine and program levels (Table 2). Substantial intercorrelations were observed among Halstead's E, McCabe's v(G), and length at the subroutine level. When computed on the total program, the correlation between length and McCabe's v(G) increased, while the correlations for Halstead's E with these two measures were substantially smaller, especially with lines of code.

Correlations between time to find the bug and the complexity metrics were calculated for unaggregated data (three experimental tasks for each of the 54 participants, $n = 162$) and for data averaged over the six scores obtained for each program (Table 3). Correlations for the aggregated data were much higher than those for the unaggregated scores. All three metrics predicted performance equally well at the subroutine level. At the program level, however, E was the best predictor, accounting for more than twice the variance in performance than did the length (56% versus 27%, respectively). The variance accounted for by v(G) fell between these values (42%). A stepwise multiple regression analysis indicated that length and v(G) added no increments to the prediction afforded by E.

The scatterplot of performance with Halstead's E presented in Figure 7 suggested the existence of a curvilinear trend in the data. The significance of this trend was tested using the second degree polynomial regression approach suggested by both Cohen and Cohen (1975) and Kerlinger and Pedhazer (1973) for investigating curvilinear relationships. A multiple correlation coefficient of 0.84 indicated that the curvilinear trend accounted for an additional 15% ($p \leq 0.001$) of the

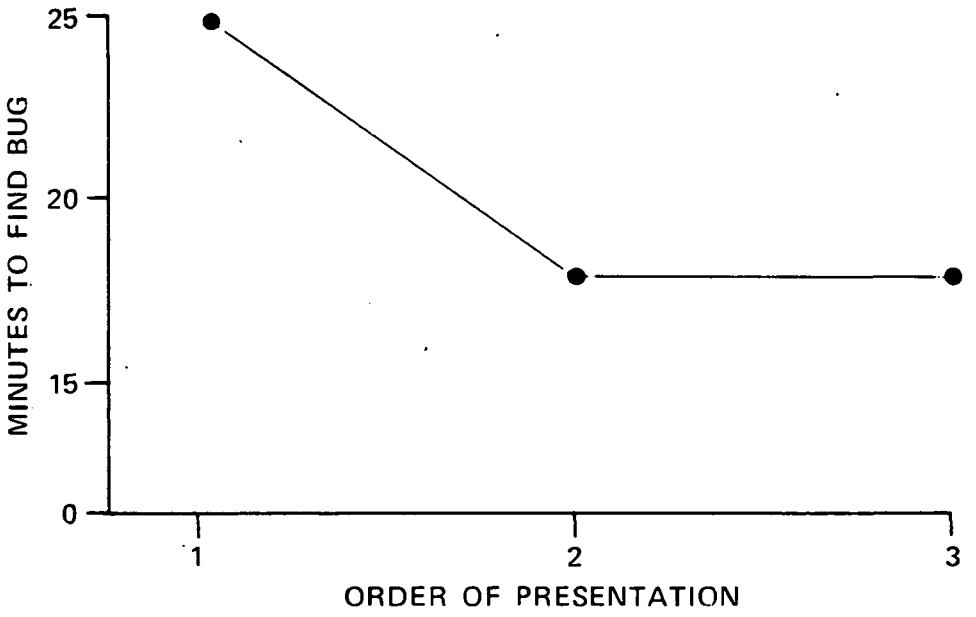


Figure 5. Order Effect on the Three Experimental Tasks

Table 1

Hierarchical Regression Analysis for Time to Find Bug

Variable	df	R^2	ΔR^2
(1) Program	2	0.06**	0.06**
(2) Presentation order	2	0.04*	0.04*
(3) Type of bug	2	0.00	0.00
(4) Program X bug interaction	4	0.26***	0.26***
(5) Complexity of control flow	2	0.02	0.02
All variables	12		0.38***

NOTE: $n = 162$. R^2 column represents the separate regression for each domain.

*$p \leqslant 0.05$
**$p \leqslant 0.01$
***$p \leqslant 0.001$

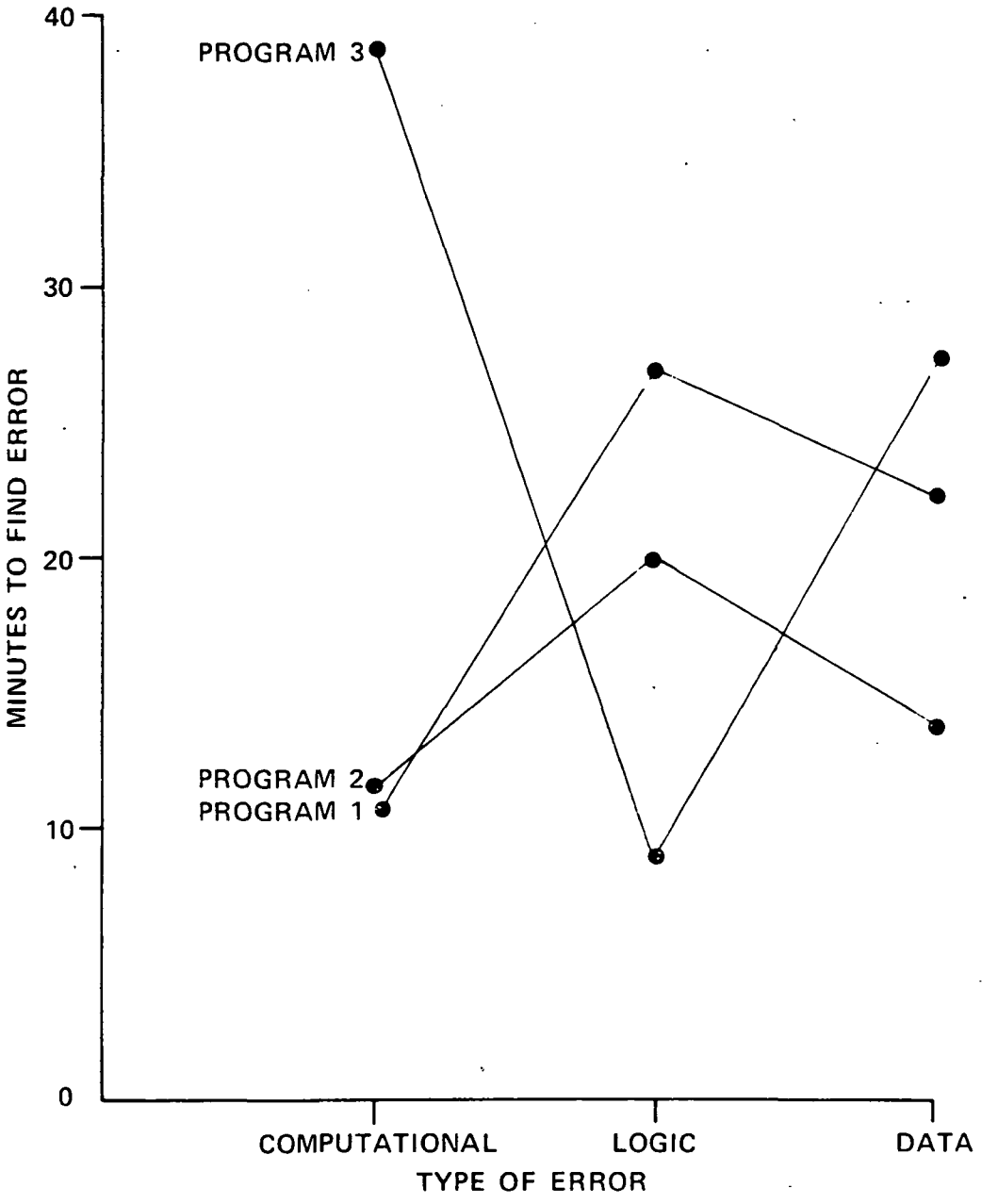


Figure 6. Program by Error Interaction

Table 2

Intercorrelations Among Complexity Metrics

Metrics	Correlations	
	E	v(G)
Subroutine:		
v(G)	0.92***	
Length	0.89***	0.81***
Program:		
v(G)	0.76***	
Length	0.56***	0.90***

NOTE: n = 27.

***p ≤ 0.001

Table 3

Correlation Between Performance Time and Complexity Metrics

Metric	Correlations	
	Unaggregated (n = 162)	Aggregated (n = 27)
Subroutine:		
Halstead's E	0.25***	0.66***
McCabe's v(G)	0.24***	0.63***
Length	0.25***	0.67***
Program:		
Halstead's E	0.28***	0.75***
McCabe's v(G)	0.25***	0.65***
Length	0.20**	0.52**

**p ≤ 0.01
***p ≤ 0.001

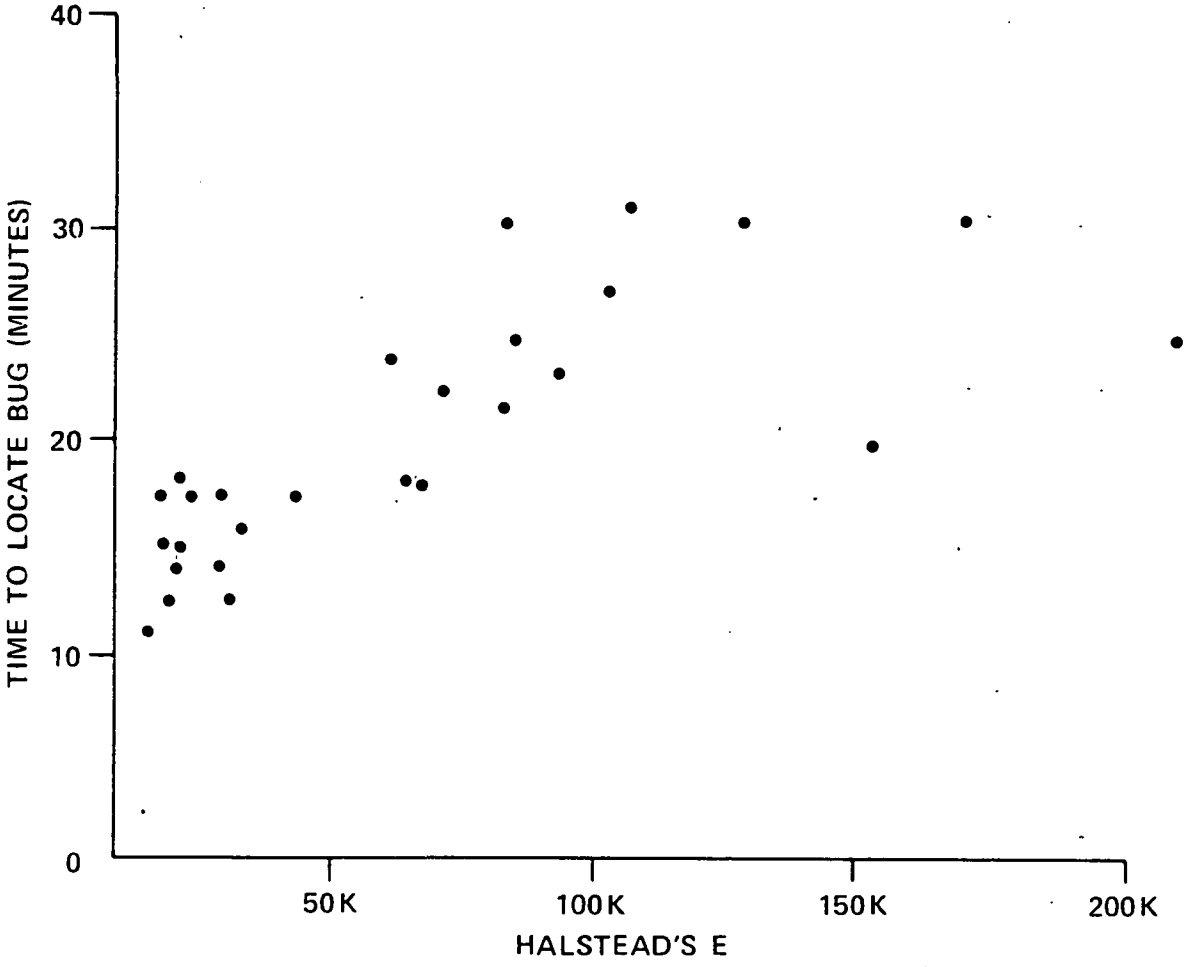


Figure 7. Scatterplot of Halstead's E and Performance

variance beyond that accounted for by a linear relationship. The prediction equation generated from these data was:

$$\text{minutes to find bug} = 9.837 + 0.00239\underline{E} - 0.00000000079\underline{E}^2$$

However, with few data points in the right tail of this distribution for Halstead's $\underline{E}$, it is difficult to extrapolate to the exact shape of the curvilinear trend. No curvilinear trend was detected with either the lines of code or McCabe's $\underline{v(G)}$.

Experiential Factors

The relationship between complexity metrics and performance was investigated within groups of programmers differing in years of professional experience programming in Fortran. As a heuristic, the participants were divided into two groups of approximately equal numbers: those with three or fewer years experience and those with more than three years experience. The results presented in Table 4 indicate that the complexity measures were more predictive of performance for less experienced programmers, especially when computed at the subroutine level.

Two measures of experience were also found to be related to the performance of less experienced programmers (Table 5), but not to the performance of experienced programmers. The first such measure was the number of programming languages the participant knew. The second metric was the number of items checked on the experience questionnaire. The moderating effects of programmer experience may have been the result of greater variability in performance for programmers with less experience (Fig. 8). This greater variability would increase the ability of correlational tests to detect significant relationships (Cohen and Cohen, 1975).

DISCUSSION

Four factors were found to influence the speed with which programmers could find a bug in a computer program. These factors were order of presentation, specific program, a program by error interaction, and the complexity of the code as measured by software complexity metrics. Type of bug and type of control flow, however, did not account for a significant proportion of the variation in performance.

Variance in programmer performance associated with differences among the programs replicated results from two previous experiments in this series (Sheppard, et al., 1979). However, a much larger percent of the variance in performance was accounted for by a program by error interaction. It appeared that some quality of the algorithm in which the bug was embedded influenced a programmer's ability to locate it. The time required to detect similar errors contained in similar statements depended on the program in which the error was embedded. This result has implications for the usefulness of various schemes for categorizing software bugs. The implied value of these taxonomies is to identify properties of bugs which suggest how they are created or how difficult they are to detect. Simple taxonomies based on syntactic relationships will probably not prove sufficient for this purpose. The results of this experiment suggest that the detectability of a bug depends on the context of the algorithm surrounding it. This contextual effect may determine the optimal search strategy for finding the bug, and it is this search strategy that needs to be understood if debugging performance is to be improved.

Table 4

Correlations Between Performance and Complexity Metrics Moderated by Years of Fortran Experience

Metrics	Correlations	
	⩽3 Years (n = 75)	>3 Years (n = 87)
Subroutines:		
Halstead's E	0.39***	0.11
McCabe's v(G)	0.37***	0.07
Length	0.33***	0.17
Program:		
Halstead's E	0.38***	0.20*
McCabe's v(G)	0.29***	0.21*
Length	0.18	0.22*

NOTE: Dividing the data into groups of programmers required that scores be analyzed on individual tasks rather than on tasks averaged by program. Thus, this analysis was performed on the 75 experimental tasks performed by the 25 participants with 3 or fewer years of Fortran experience and the 87 tasks performed by the 29 participants with more than 3 years experience.

*$p \leqslant 0.05$
**$p \leqslant 0.01$
***$p \leqslant 0.001$

Table 5

Relationships of Experiential Factors to Performance for Programmers Differing in Fortran Experience

Relevant Experience	⩽3 Years (n = 25)	>3 Years (n = 29)	Total (n = 54)
# of Programming Languages	-0.49**	-0.03	-0.19
Questionnaire Score	-0.48**	-0.11	-0.33**

**$p \leqslant 0.01$

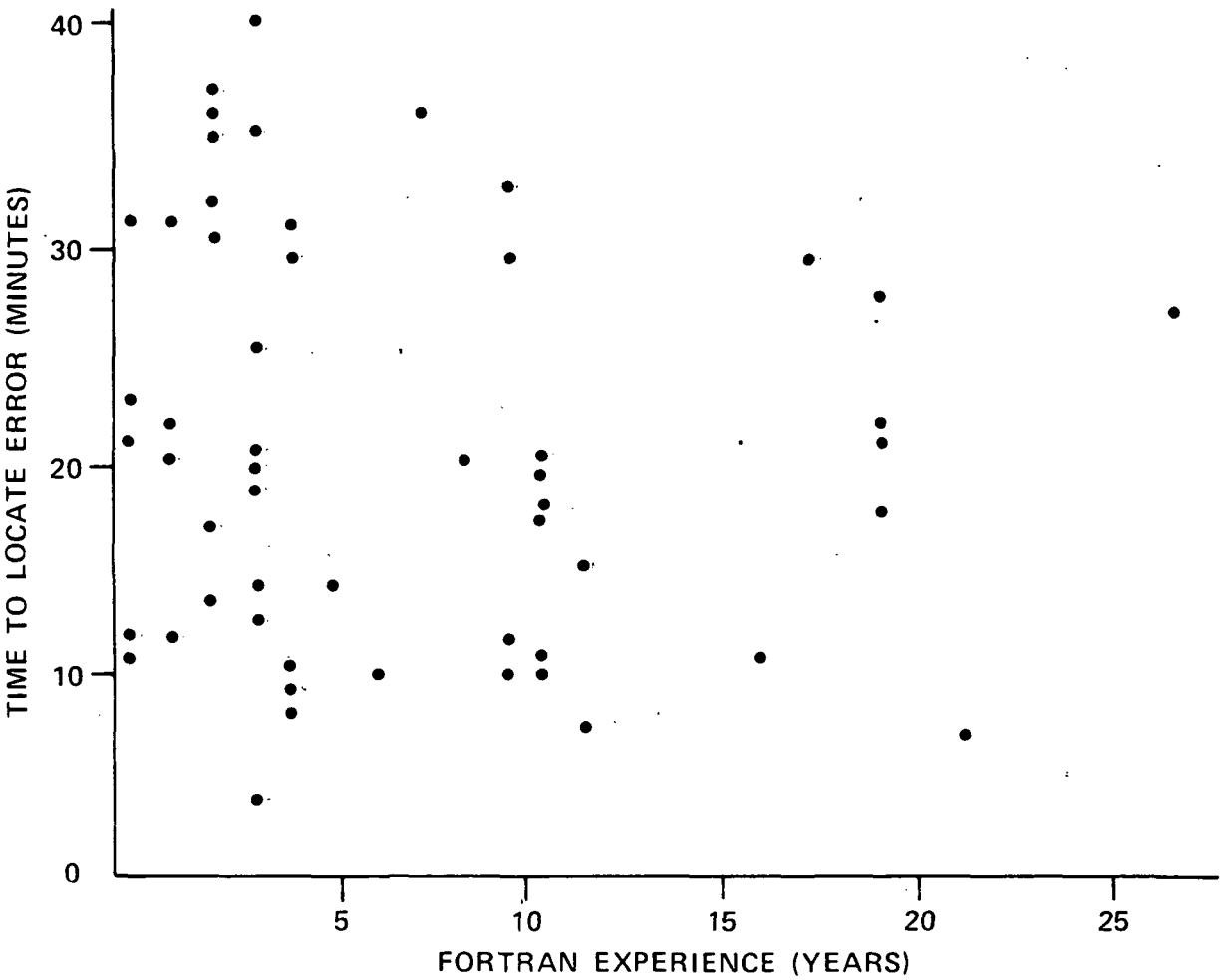


Figure 8. Scatterplot of Experience and Performance

In the last section of the post-session questionnaire, the participants were asked to describe their searching strategies for locating the bugs. Typically, one of two approaches was described. In the first strategy the programmer tried to understand the whole program from beginning to end before searching for the section with the bug. In the second strategy the programmer used appropriate clues in the output to go directly to the section containing the bug. The latter appeared to be a much quicker strategy for debugging, but there were insufficient data for a meaningful statistical analysis. In order to improve the debugging performance of programmers it will be important not only to identify effective search strategies, but also to identify conditions under which they will be differentially effective.

No significant differences were evident among the three types of top-down control flow tested in this experiment. This finding agrees with previous results (Sheppard, et al., 1979) where differences were found between top-down and convoluted control flow, but not between types of top-down control flow. The minor deviations from strictly structured coding allowed in the naturally structured version of this experiment did not adversely affect performance. Summarizing the combined results of the three experiments, it would appear that the overall top-down quality of the control flow is important to performance, but careful attention to strict structuring does not appear to improve programmer performance significantly.

Since no difference was found between the graph-structured and Fortran 77 program versions, it would appear that the newer constructs provide little additional aid in a debugging task beyond that provided by a top-down flow. Only five of the 54 participants had previously used Fortran 77, so a lack of familiarity with the new constructs may have prevented them from finding the bug more quickly in Fortran 77 than in Fortran IV. However, immediately prior to the experiment a short training session was conducted with each group of participants in which the new Fortran 77 constructs were discussed in detail. These constructs were similar to those implemented in Fortran IV, and the participants' previous lack of familiarity with them was probably not a significant factor in their performance.

Most laboratory studies exhibit a certain degree of artificiality that is necessary for experimental control. In this experiment participants were told there was only one bug in a program. While this situation differs from a normal programming environment, it should not have affected participant's ability to perform the tasks. These experimental tasks may have been simpler to perform than typical debugging problems since there was greater certainty about the bugs. Further, differences between the correct and erroneous output were clearly marked on the erroneous output, reducing the amount of comparison necessary to discover what problems had occurred.

During a typical debugging problem a programmer could refer to the functional specifications for a program or to comments included in the code. However, no such aids were made available in this experiment. The participant's comprehension of the program's function had to be gleaned from the code or from the input and output listings. The latter were designed to be self-explanatory, with each section labeled appropriately; e.g., "FINAL COURSE GRADE" or "TRIAL BALANCE." Although adding some artificiality to the experimental situation, the absence of documentation was an attempt to equalize the amount of information provided by materials other than the code.

Software Complexity Metrics

The results of this experiment not only replicated the results obtained in our previous research, but also demonstrated that more viable results could be obtained when limitations in our earlier

experimental procedures were overcome. For instance, our previous research was conducted exclusively on small-sized (35–55 lines of code) programs, which seems to have limited the results in three ways. First, the range of values on the factors studied in those programs seems to have been too restricted to detect the size of relationships observed here. Second, the curvilinear relationship observed in this experiment between Halstead's E and performance would not have been observed if longer programs had not been used in the experimental tasks. Third, the extremely high intercorrelation between length and Halstead's E at the subroutine level suggests that both are measuring program volume. With larger programs the information measured appears to differ; that is, Halstead's E measures something in addition to, but inclusive of, factors measured by length.

Many small-sized programs can be grasped by the typical programmer as a cognitive gestalt. The psychological complexity of such programs is adequately represented by the volume of the program in terms of the number of lines of code. When the code grows beyond a subroutine, its complexity to the programmer is better assessed by measuring constructs other than the number of lines of code. This may result partly because programmers cannot grasp the entire program within their mental spans at a single time. For larger programs the difficulty programmers experience is better represented by counts of operators, operands, and control paths. Thus, as the size of a program increases, Halstead's E seems to be a better measure of its psychological complexity.

One possible explanation for the superior predictive ability of Halstead's E is that the relationship between program size and performance is curvilinear, and the algorithmic transformation with the Halstead measure captures this relationship while lines of code does not. There was no evidence in these data of a curvilinear relationship between lines of code and performance. On the other hand, a curvilinear relationship did exist between Halstead's E and performance. This trend suggests that as Halstead's E grows larger, a program becomes more psychologically complex, but the increments in difficulty grow smaller and smaller. In the experimental task used in this debugging experiment, there seemed to be an amount of time that was typically required to locate a bug within a subroutine once the correct subroutine had been identified (approximately 16 minutes). Added to this baseline rate was the time required to identify the proper subroutine. The curvilinearity of the relationship between time to find the bug and Halstead's E appeared to result from the time required to isolate the problem subroutine.

The moderating effects of experiential factors also replicated the results found in the earlier experiments. The metrics again proved to be better predictors of performance for programmers with three or fewer years experience in Fortran than for those with more than three years experience. It was also possible to predict the performance of an individual programmer from job history data. Several important factors seemed to be the number of languages a programmer had used and familiarity with certain programming concepts. These predictions from job history were also more valid for programmers who had three or fewer years of experience in Fortran. Future work is needed to refine the use of experiential questionnaires for use in personnel functions such as selection, assessment for training needs, and placement.

Code which is more psychologically complex may also be more error-prone and difficult to test. The results of this experiment provide evidence that the software complexity metrics developed by Halstead and McCabe are related to the difficulty programmers experience in locating errors in code. Thus these metrics appear to be capable of satisfying several practical applications. They can be used in providing feedback both to programmers about the complexity of the code they

have developed and to managers about the resources that will be necessary to maintain particular sections of code. Further evaluative research needs to assess the validity of these uses in ongoing software projects.

ACKNOWLEDGEMENTS

The authors are grateful to Judy McWilliams and Mary Anne Borst who helped with this experiment and to Beverly Day for manuscript preparation. We are also grateful to Dr. Gerald Hahn for advice on experimental design, to Drs. Tom Love and Ben Shneiderman for advice on the experimental tasks and procedures, and to Dr. John O'Hare for his careful review of this report. We are especially appreciative of the efforts of Earl North and Leo Pompliano of General Electric; Jan Gombert of Applied Urbanetics; Mrs. Joan Shields, Cols. William Eglington, Earl Goetze and Richard Blair, and Lt. Col. Pat Harris of the U.S. Air Force; and Capt. Webster and J. Rehbehn of the U.S. Navy in providing the participants for this research. The support and encouragement of both Gerald Dwyer and Lou Oliver has been vital to the success of this research.

This research was supported by the Office of Naval Research, Engineering Psychology Programs (Contract #N0014-77-C-0158). The views expressed in this paper, however, are not necessarily those of the Office of Naval Research or the Department of Defense.

REFERENCES

Brainerd, W., Fortran 77. Communications of the ACM. 1978, 21, 806-820.

Brooks, R. Unpublished algorithm. Irvine, CA: University of California at Irvine, Computer Science Department, 1978.

Campbell, D. and J. C. Stanely, Experimental and quasi-experimental designs for research. Chicago: Rand-McNally, 1967.

Carlson, W. E. and B. DeRoze, Defense system software research and development plan. Unpublished manuscript, Arlington, VA: Defnese Advanced Research Projects Agency, September 1977.

Cohen, J. and P. Cohen, Applied multiple regression/correlation analysis for the behavioral sciences. New York: Wiley, 1975.

Curtis, B., S. B. Sheppard, P. Milliman, M. A. Borst, and T. Love, Measuring the psychological complexity of software maintenance tasks with the Halstead and McCabe metrics. IEEE Transactions on Software Engineering, 1979, 5, 95-104.

Department of Defense requirements for high order computer programming languages: Revised "IRONMAN." SIGPLAN Notices, 1977, 12, 39-54.

DeRoze, B., Software research and development technology in the Department of Defense. Paper presented at the AIIE Conference on Software, Washington, D.C.: December 1977.

Dijkstra, E. W., Notes on structured programming. In Structured programming, O. J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, (Ed.) New York: Academic, 1972.

Fitzsimmons, A. B. and L. T. Love, A review and evaluation of software science. ACM Computing Survey, 1978, 10, 3-18.

Gordon, R. D., A measure of mental effort related to program clarity. Unpublished doctoral dissertation, Purdue University, 1977.

Gould, J. D., Some psychological evidence on how people debug computer programs. International Journal of Man-Machine Studies, 1975, 7, 151-182.

Gould, J. D. and P. Drongowski, An exploratory study of computer program debugging. Human Factors, 1974, 16, 258-277.

Halstead, M. H., Elements of software science. New York: Elseiver North-Holland, 1977.

Hecht, H., W. A. Sturm, and S. Trattner, Reliability measurement during software development. Redondo Beach, CA: Aerospace Corp., 1978.

Kerlinger, F. N. and E. J. Pedhazur, Multiple regression in behavioral research. New York: Holt, Rinehart, and Winston, 1973.

McCabe, T. J., A complexity measure. IEEE Transactions on Software Engineering, 1976, 2, 308-320.

Nolen, R. L., Fortran IV computing and applications. Reading, MA: Addison-Wesley, 1971.

Ottenstein, K. J., A program to count operators and operands for ANSI-FORTRAN modules (Tech. Rep. CSD-TR-196). West Lafayette, IN: Purdue University, Computer Science Department, 1976.

Sheppard, S. B., B. Curtis, M. A. Borst, P. Milliman, and L. T. Love, First year results from a research program on human factors in software engineering. In Proceedings of the 1979 National Computer Conference, Montvale, NJ: AFIPS, 1979.

Sheppard, S. B. and L. T. Love, A preliminary experiment to test influences on human understanding of software. In Proceedings of the 21st Meeting of the Human Factors Society. Santa Monica, CA: Human Factors Society, 1977.

Tenny, T., Structured programming in FORTRAN. Datamation, 1974, 20, 110-115.

The military software market (Rep. 427). New York: Frost and Sullivan, 1977.

Veldman, D. J., Fortran programming for the behavioral sciences. New York: Holt, Rinehart, and Winston, 1967.

Wescourt, K. T. and L. Hemphill, Representing and teaching knowledge for troubleshooting/ debugging (Tech. Rep. 292). Stanford, CA: Stanford University, Institute for Mathematical Studies in Social Science, 1978.

Youngs, E. A., Human errors in programming. International Journal of Man-Machine Studies, 1974, 6, 361-376.

APPENDIX A

PRETEST

Sorting Algorithm

INPUT

DATAPRE

```
 25
110
 30
 31
  1
153
193
 62
 78
 16
  1
193
 62
 78
 74
168
192
199
999
  5
 78
 79
 56
  9
 57
  3
```

```
100         IMPLICIT INTEGER(A-Z)
110         DIMENSION A(50),B(50)
115         READ('DATAPRE',10) N
116         DO 5 I = 1, N
120     5   READ('DATAPRE',10) A(I)
130    10   FORMAT(I3)
140         DO 100 J = 1, N
160         SMALL = A(1)
170         M = 1
180         DO 20 K = 2,N
190    15   IF(A(K) .LT. SMALL) GO TO 20
200         SMALL = A(K)
210         M = K
220    20   CONTINUE
230         B(J) = SMALL
240         A(M) = 1000
250   100   CONTINUE
251         DO 101 I = 1, N
260   101   PRINT 110, B(I)
261   110   FORMAT(2X,I4)
270         STOP
280         END
```

CORRECT OUTPUT	INCORRECT OUTPUT
1	999
1	1000
3	1000
5	1000
9	1000
16	1000
30	1000
31	1000
56	1000
57	1000
62	1000
62	1000
74	1000
78	1000
78	1000
78	1000
79	1000
110	1000
153	1000
168	1000
192	1000
193	1000
193	1000
199	1000
999	1000

S. Sheppard
G.E.

APPENDIX A

PRETEST

INTRODUCTION

DDI is a software development company currently in charge of maintaining the Advanced Orbit Ephemeris Subsystem (AOES) for the USAF. This presentation will address the various methods used in maintaining and upgrading the AOES, and to show how these methods reduce the number of discrepancies in the AOES.

MAINTAINING THE EXISTING SYSTEM

Requirements are generated by the user community to either modify or upgrade the current AOES. These requirements can modify existing programs or create programs which are then added to the AOES. The development of these requirements into software programs are delivered to the Air Force on a scheduled date and this delivery is called a MODEL. Any discrepancies found in the current or past models are corrected using machine code. The machine code is later converted to a HOL (JOVIAL) for incorporation into a future model.

PROBLEMS FACED IN MAINTENANCE

The original programs were all written in a very non-structured manner. The program logic in most of the original programs are very difficult to follow since more than one programmer was involved in the original coding and subsequent modifications. The comments are obscure, non-meaningful or absent in some instances. Furthermore, many discrepancies are corrected without any thought to future modifications in the area of the fix or to the readability of the correction (i.e., the correction seems to appear out of place in the area in which it occurs).

It mentioned in the introduction the objective of DDI was to try to alleviate discrepancies against delivered models. To do this Structured Software Techniques, and the formation of a Quality Assurance staff was implemented.

This presentation will describe these tools and their effectiveness and their weakness as they have been observed.

STRUCTURED SOFTWARE TECHNIQUES

- CURRENT METHODS
  - Top Down Design
    - Software Engineering, group leader and programmer sit down and review the requirement(s) for a new software program or for modifications to existing software programs. All major areas of the requirements are identified, and

these are further subdivided into lesser tasks. This process is repeated until each task can be dealt with separately.

- Effectiveness
  - Early in the development cycle all major interfaces for the requirements can be identified.
  - Any design trade–offs will surface and can be further analyzed.
  - Having the programmers involved gives them a better understanding of the possible problem areas and greater involvement in the final delivered product.

- STRUCTURED PROGRAMMING TECHNIQUES
  - The higher order language that is used by DDI is called JOVIAL. This language is very readable, flexible and very well suited for structured programming. The only constructs missing to truly make it an ideal structured langugae are the DO WHILE and CASE instructions.
  - The following guidelines are followed in the modification or development of software.
    - Comments
      - Have meaningful comments.
      - A comment should appear in at least every 3 or 4 lines of JOVIAL code, for every conditional statement and block structure.
      - Comment all items, arrays and tables.
    - All Items, Variables, Arrays and Tables
      - Alphabetized within their respective groups.
      - Distinct and have meaningful names.
      - Start them all in Column 4.
      - Overlays and defines are to be at the end of the parameter list.
      - Table entries should follow the indentation rules. Also, the presets of of tables and arrays.

- JOVIAL Executable Statements
  - Start in Column 4.
  - Are assigned to one line, and if more than one line is required, indent the continuation line at least 3 columns.
  - GOTO statements should be used with discretion.
- Conditional Statements and Block Structures
  - Indent statements following conditional statements by a minimum of 3 columns.
  - Indent block structure by 3 columns and identify the begin and end of each block.
- PROCS and CLOSES (Internal subroutines)
  - Whenever there is a choice use PROCS.
  - Start the Statement PROC or CLOSE in Column 1.
  - Whenever feasible try to pass single input and single output parameter.
  - Do not use the same input and output names in several PROCS.
  - Attempt to alphabetize your PROCS at the end of your program.
  - JOVIAL code, ITEMS, tables and arrays should start in Column 4.
  - For each PROC or CLOSE describe its purpose and all of its input and output parameters.
- Statement Labels
  - Start in Column 1.
  - Be assigned an individual line.
  - Have descriptive names.
  - For those in PROCs or CLOSEs the first few characters of the name can be used within that PROC or CLOSE.

- Effectiveness
  - Typographically, the program is more readable.
  - Programs are more readily understood.
  - Debugging and maintenance is greatly simplified.
  - Modifications can be more easily performed.

- STRUCTURED WALK-THROUGHS
  - After the programmer has coded his program a walk through of the code is performed between the programmer and the respective group leader.
  - *After the first clean compilation another program walk-through is exercised.*
  - During the final check-out phase a final walk-through is performed.
  - A walk-through of the developmental test deck is also performed to insure that the programmer test methods do indeed test those requirements and their interfaces of the program.
  - Effectiveness
    - To insure that the programmer has coded to meet the requirements.
    - Provide a check to determine if structured software guidelines are being performed.
    - Final walk-through is an insurance step to determine if any code change has affected meeting software requirements.
    - Development test deck walk-throughs insure more discrete or better testing methods by collapsing or expanding certain tests, or by adding new tests.

- PROGRAMMER NOTEBOOK
  - This is a text of information given to, created by or used by the programmer in developing programs for a development cycle. The contents include:
    - Schedules
    - Requirements
    - All design modifications
    - Initial data flow

- All documentation and their review comments and responses
- Any conversations concerning their program with outside agencies
- Data of program walk-throughs.

- Effectiveness
  - The programmer, group leader, software engineer or project director can assess materials used in the development of each program.
  - Historical records provide insight into an individual's thoughts and logic.
  - Programmers can refer to the notebook for insight for future modifications to the same program.
  - Especially useful if an individual leaves in the middle of the development and another individual must finish the development.

- TOP DOWN TESTING
  - This is the method of testing of all top level program modules before lower level modules are tested. Top down testing allows the testing of major interfaces first. Coding for a program need not be complete before top down testing can start, since stubs can be used.
  - Not all testing is done in a top down manner, in particular instances where a lower module performs some critical processing that is required at the upper levels, those lower programs are tested first using a driver program. But once those lower level programs have been tested, top down testing resumes.
  - Effectiveness
    - Both coding and testing can occur at the same time, and this leads to a better distribution of testing time.
    - Eliminates the need for driver programs to be written in order to check out the actual program.

METHODS TRIED BUT NON-EFFECTIVE FOR OUR WORK

Pseudo Code or Program Design Language

JOVIAL language can be used as a program design language and many programmers were getting too detail oriented and not looking at the structure of the program.

Flow Charting

Again, flow charting made the programmers detail oriented and not structure oriented. Flow block diagrams were only major blocks and decisions proved to be much more effective.

SOFTWARE QUALITY ASSURANCE GROUP

A software quality assurance (QA) group was created to formally validate the requirements of a model. The QA staff is a separate group of individuals whose task is to support the software development of the model. This is achieved by having a member of the QA staff sit in when the top-down design of a program is being done. This will aid the QA member to understand the requirements of the program. This understanding will be used in developing a formal system level validation test of the requirements. The QA staff will be responsible to execute all of their validation tests to verify that the user requirements have been satisfied. The QA staff has the responsibility to review all formal documentation produced by the programmers to insure that all requirements have been addressed and that the document conforms to the proper format. The QA group will be the configuration control point for each model.

Effectiveness

- Formal validation of the software requirements are centralized in a single document.
- Independent testing of software programs before a formal release.
- All discrepancies found can be more easily duplicated and solved by the programmers using a HOL.
- Configuration management control.

SUMMARY

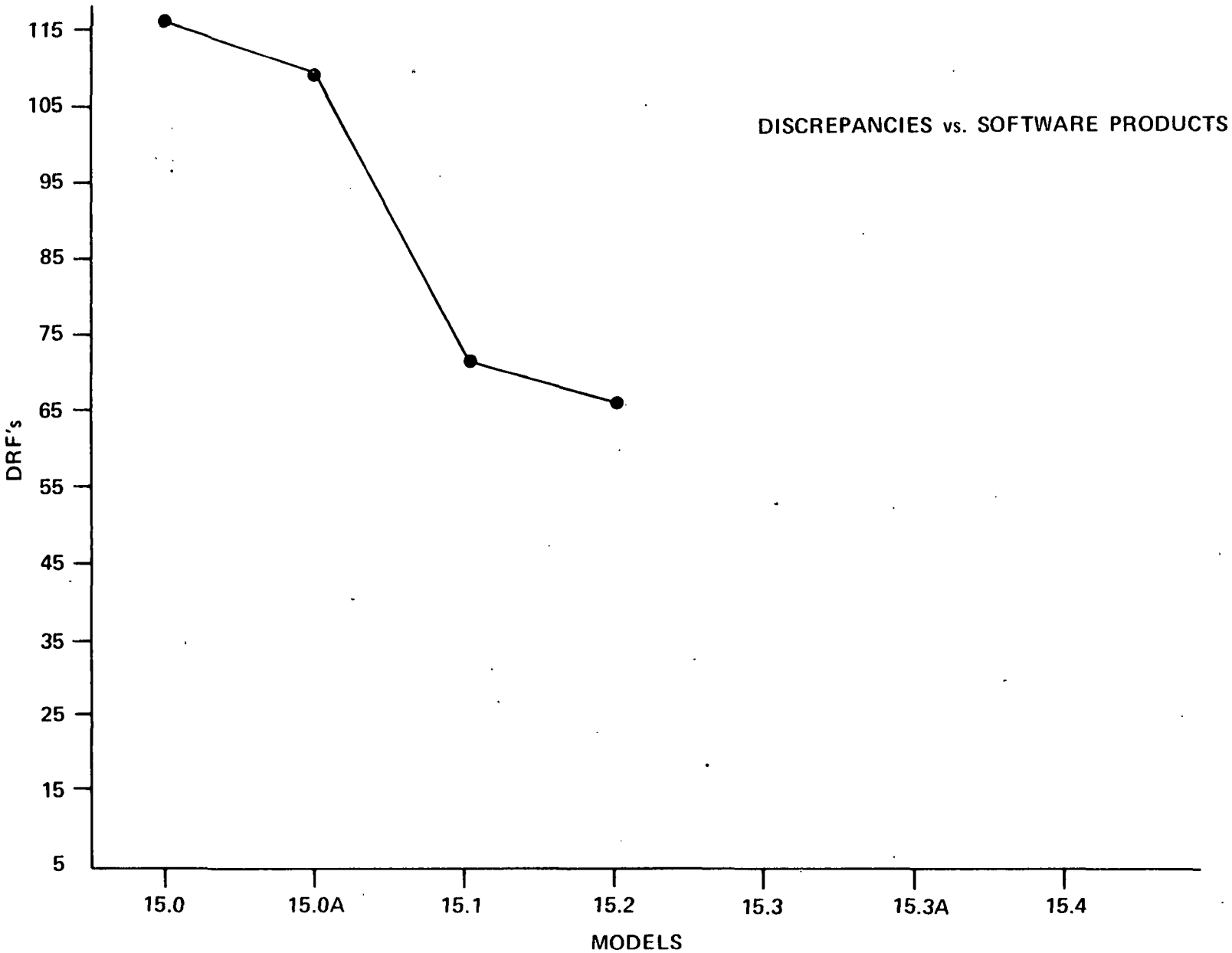
Using certain structured techniques with the added independent testing performed by the QA staff, DDI has reduced the number of discrepancies in modifying or upgrading our current system. There is a very definite advantage to applying these techniques to existing systems.

- MAINTAINING THE EXISTING SYSTEM
  - MODIFICATION TO EXISTING SOFTWARE
  - DEVELOPMENT OF NEW SOFTWARE TO AUGMENT THE CURRENT SYSTEM
  - CORRECT DESCREPANCIES FOUND IN THE CURRENT SYSTEM
- PROBLEMS FACED IN MAINTENANCE
  - ALL ORIGINAL PROGRAMS WRITTEN WITHOUT STRUCTURED TECHNIQUES
  - PROGRAM LOGIC IS DIFFICULT TO FOLLOW
  - OBSCURE COMMENTS OR NO COMMENTS
  - PATCHED AREAS
- STRUCTURED SOFTWARE TECHNIQUES
  - TOP DOWN DESIGN
    - MAJOR AREAS ARE IDENTIFIED
    - EFFECTIVENESS
      - OVERVIEW OF THE PROGRAM STRUCTURE
      - INTERFACES CAN BE IDENTIFIED EARLY
      - DESIGN TRADE-OFFS SURFACE
      - EARLY PROGRAMMER INVOLVEMENT
    - DRAWBACK
      - TOO MUCH MODULARIZATION

- STRUCTURED SOFTWARE TECHNIQUES
  - STRUCTURED PROGRAMMING TECHNIQUES
    - HOL - JOVIAL
      - COMMENTS ARE TO BE MEANINGFUL AND PLENTIFUL
      - INDENTATION OF CODE FOR CONDITIONAL STATEMENTS AND BLOCK STRUCTURES
      - MEANINGFUL NAMES FOR STATEMENT LABELS, INTERNAL SUBROUTINES, AND VARIABLES
    - EFFECTIVENESS
      - READABLE PROGRAMS
      - PROGRAM LOGIC MORE READILY UNDERSTOOD
      - DEBUGGING AND MAINTENANCE SIMPLIFIED
      - MODIFICATIONS MORE EASILY PERFORMED
    - DRAWBACKS
      - SYSTEM AND CORE LIMITATION
      - TIMING REQUIREMENTS

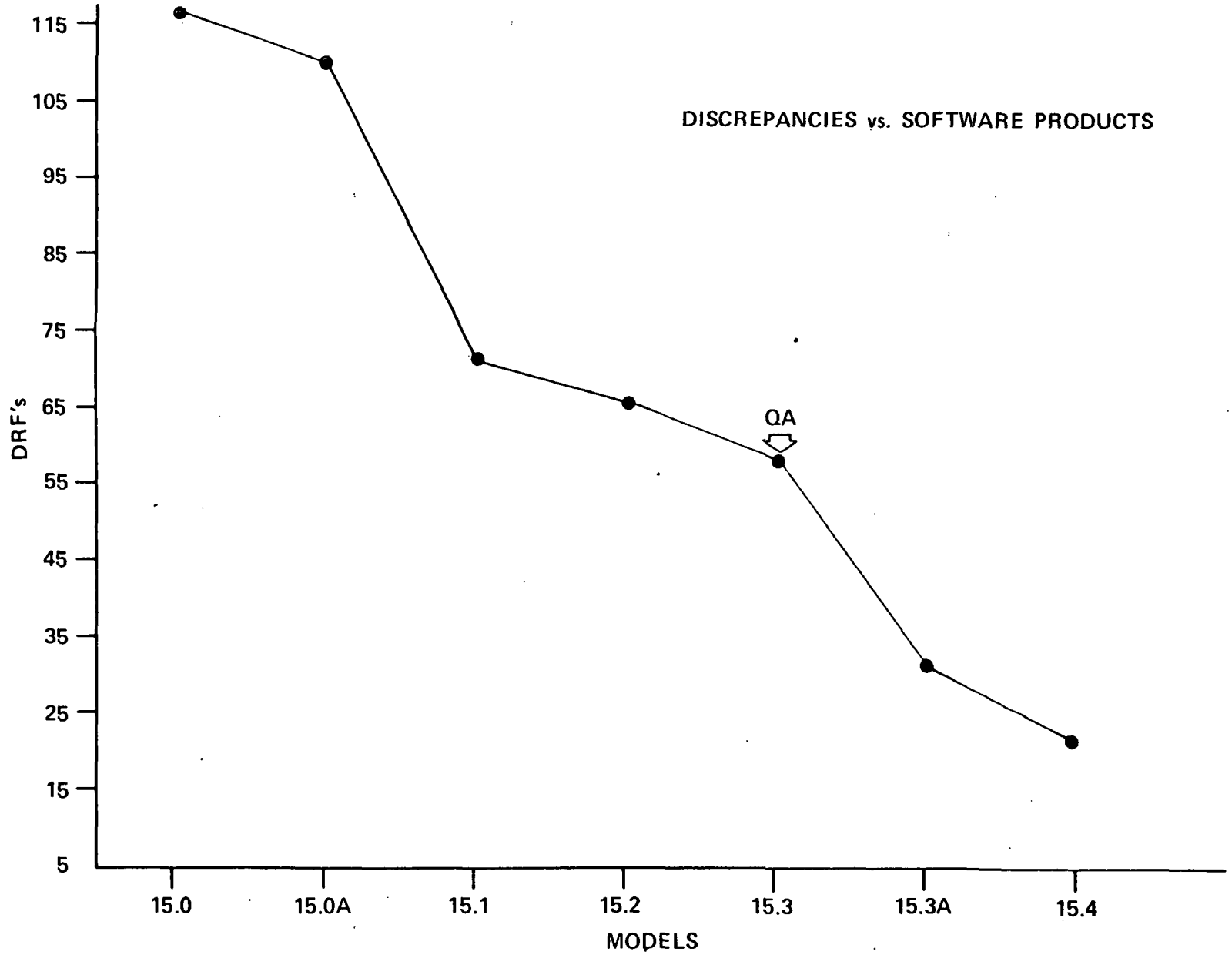
- STRUCTURED SOFTWARE TECHNIQUES
  - STRUCTURED WALK-THROUGHS
    - PROGRAM WALK-THROUGHS
      - AT LEAST THREE TIMES
    - DEVELOPMENT TEST DECK WALK-THROUGH
    - EFFECTIVENESS
      - PROGRAMMER HAS CODE TO MEET REQUIREMENTS
      - TESTING OF CODE WHICH SATISFY REQUIREMENTS
  - PROGRAMMER NOTEBOOK
    - TEXT OF INFORMATION USED TO SATISFY REQUIREMENTS
    - EFFECTIVENESS
      - HISTORICAL ACCOUNT OF PROGRAM DEVELOPMENT
      - USEFUL FOR SUBSEQUENT WORK ON THE SAME PROGRAM
      - USEFUL IF PROGRAMMER LEAVES BEFORE COMPLETION
    - DRAWBACK
      - PROGRAMMERS DO NOT ALWAYS UPDATE

- STRUCTURED SOFTWARE TECHNIQUES
  - TOP DOWN TESTING
    - USE TOP LEVEL MODULES TO TEST LOWER LEVEL MODULES
  - EFFECTIVENESS
    - MAJOR INTERFACES ARE TESTED FIRST
    - CODING DOES NOT HAVE TO BE COMPLETE, USE OF STUBS
    - ELIMINATION OF DRIVER PROGRAMS
    - BETTER DISTRIBUTION OF TESTING TIME
  - DRAWBACK
    - NOT ALL TESTING CAN BE DONE TOP DOWN


DISCREPANCIES vs. SOFTWARE PRODUCTS
DRF's
115
105
95
85
75
65
55
45
35
25
15
5
15.0
15.0A
15.1
15.2
15.3
15.3A
15.4
MODELS

- SOFTWARE QUALITY ASSURANCE (QA)
  - SUPPORT SOFTWARE DEVELOPMENT
    - UNDERSTAND REQUIREMENTS
    - FORMAL VALIDATION OF SOFTWARE REQUIREMENTS USING SYSTEM LEVEL TESTING
    - REVIEW DOCUMENTATION
    - CONFIGURATION CONTROL
  - EFFECTIVENESS
    - FORMAL TESTING IS CENTRALIZED
    - INDEPENDENT TEST
    - MINIMIZE DELIVERY PROBLEMS

W. Fujii
DDI


DISCREPANCIES vs. SOFTWARE PRODUCTS
DRF's
115
105
95
85
75
65
55
45
35
25
15
5
QA
15.0
15.0A
15.1
15.2
15.3
15.3A
15.4
MODELS

PANEL #4

SOFTWARE RESOURCE MODELS

B. Cheadle, Martin Marietta
L. Putnam, Quantitative Software Management
D. Weiss, NRL

SOFTWARE RESOURCE MODELS PANEL #4

William G. Cheadle
Martin Marietta Aerospace
P.O. Box 179, S-2530
Denver, Colorado 80201

ABSTRACT

My 10-15 minutes will be spent on discussing the importance of understanding the Software Development Process and using this knowledge when applying software estimating models.

- Where does Successful Software Development begin? We will answer this question.
- What is Successful Software Development? We will provide an answer.
- A good Software Estimate requires that several things be accomplished during the Planning Phase prior to Contract award; like functional decomposition, software sizing, identification of programming languages, and identifying complexities to come up with an estimate of Software Resource using a software model and data base.
- We will discuss the importance of creating a Software Development Plan (SDP) during the Planning Phase which identifies the Software Development Phase, Subphases, Design Reviews, documentation, and how we plan to test the software.
- Software Development Schedules will be talked about and how to manload the Software Development effort.
- Each organization that develops software should have the necessary tools, people, and methodology to allow them to accomplish the necessary tasks to identify required software resources.

SOFTWARE SIZING, ESTIMATING AND SCHEDULING

Where Does Successful Software Development Begin?

- In the planning phase – prior to contract award.

Why?

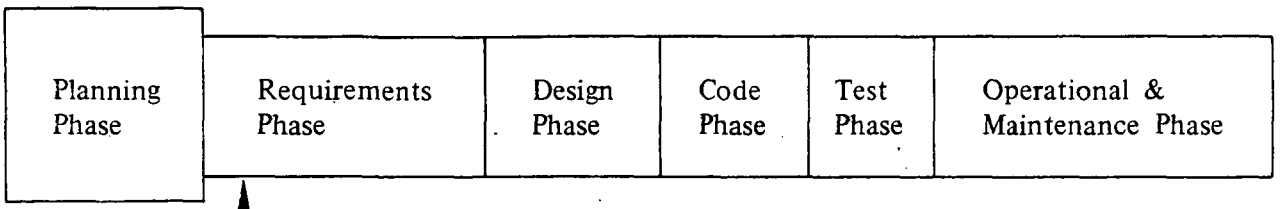
- To allow the contractor to make an achievable and competitive response to the customer's S.O.W. In other words to make a good competitive software estimate.

What Is Successful Software Development?

- Satisfying customer needs
- Staying within costs
- Meeting schedules
- Making a profit

A detailed understanding of customer requirements, and the Software Development process along with a good accurate sizing method is 75% of the estimating task.

To make a good software estimate requires accomplishing several things during the planning phase.



- We must understand the problem.
  - Customer RFP
  - Customer SOW
  - Customer needs
  - How we plan to solve customer problems
  - Class of Software
- We must understand what products are to be delivered:

  Requirements definition is a must.
  - Customer requirements
  - Derived requirements
  - Operational requirement priority list
- We must identify how we plan on developing the software by producing a good Software Development Plan (SDP). This Software Development Plan describes the Phases, Subphases, Design Reviews, documentation and how we plan to test the software.

W. Cheadle
Martin Marietta Aerospace

PHASE AND SUB-PHASES OF SOFTWARE DEVELOPMENT

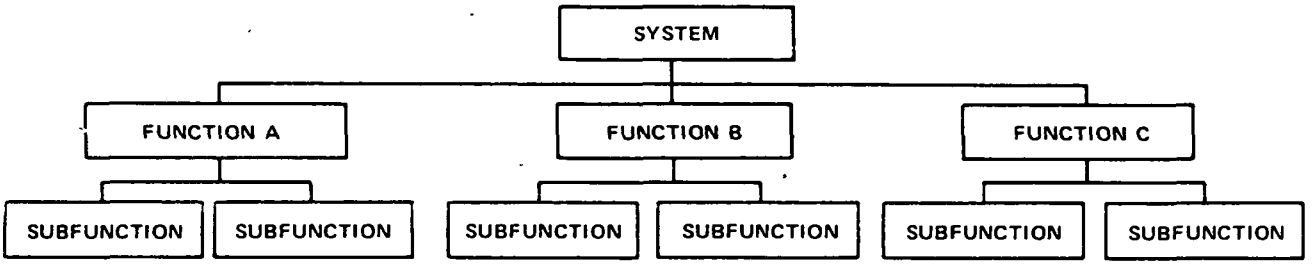
SRR △	SDR △	PDR △	CDR △	TRR △	FCA △	PCA △	FQR △

Requirements			Design				Test			Operational and Maintenance Phase
Sys Reqts	Sys Alloca-tion	S/W Reqts	Prel Design	Detail Design	Code	Check-Out	Unit PQT	Integra-tion PQT	System	

SRR Systems Reqts Review
SDR Systems Design Review
PDR Preliminary Design Review
CDR Critical Design Review
TRR Test Readiness Review
FCA Functional Configuration Audit
PCA Physical Configuration Audit
FQR Formal Qualification Review

In the SDP we must:

- Identify required documentation and when it will be produced and reviewed.
- During the Planning Phase we must accomplish some top level functional decomposition to aid in sizing the software.

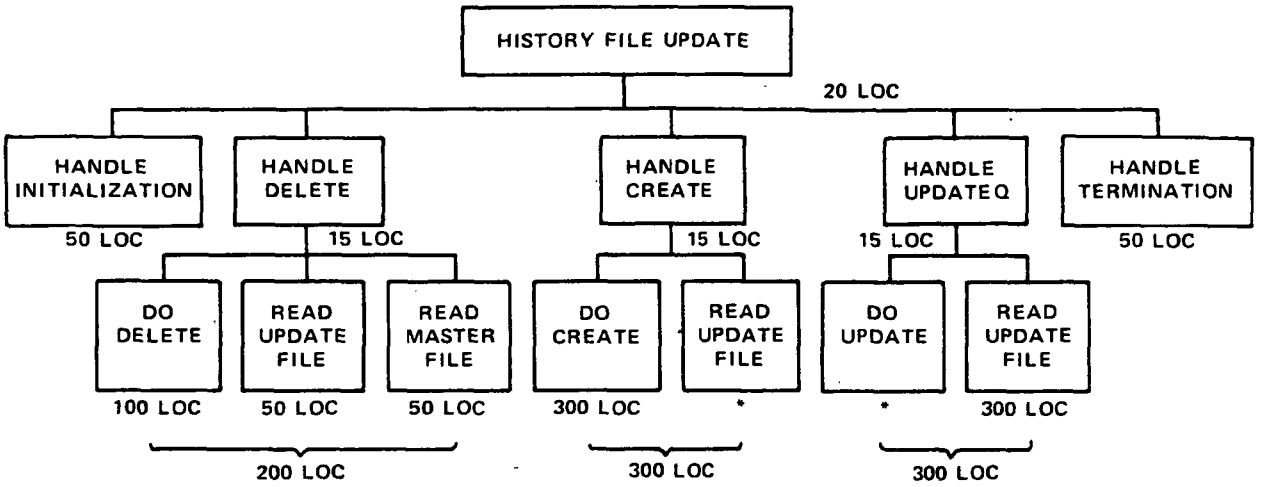


- We must size identified functions (modules)
  Sizing in source code by type of software
  Type of Software
  { Systems
  { Applications
  { Support
- We have to identify the programming language
  { Machine Language
  { Assembly Language
  { HOL

During the Planning Phase we must accomplish in a top level manner all the activities through PDR.

W. Cheadle
Martin Marietta Aerospace

EXAMPLE OF FUNCTIONAL DECOMPOSITION



*Existing Redundant Code

Language FORTRAN HOL

Type S/W

System	0
Application	965
Support	0
	965

Rules for Counting Source Code

We count executable delivered lines including data declarations but we do not count comments.

Operating System – must be identified

Support Software – must be identified

- Project Complexity must be identified:
- Software mix complexity must be identified:
- We must identify time frame required for the software development effort { Compare our Schedule with SOW schedule

Once we have completed sizing, and determined complexities we can make estimates:

- Manmonth Estimate for Software Development Phase is made:
- Estimate is made for the operational phase "In Scope" maintenance effort.
- We use our model to determine time frame required for Software Development.
- Software Development Computer Costs are identified in an estimate.

W. Cheadle
Martin Marietta Aerospace

Example:

Software Estimate for "History File Update"

Project Complexity Average

Mix Complexity Simple to Average

Source Loc 965 HOL (Fortran)

Complexity and language factors are applied for Applications Software using our software estimating model to determine budget required for Software Development.

BUDGET FOR SOFTWARE DEVELOPMENT AFTER APPLYING THE FACTORS IS:

7.8M/M

The Budget is spread across phases and subphases of Software Development:

Requirements			Design		Code		Test			Operational Maintenance Phase
Sys Reqts	Sys Allocation	S/W Reqts	Prel Design	Detail Design	Code	Check-Out	Unit	Integration	System	
x	x	x	x	x	x	x	x	x	x	Total 7.8M/M
1.8			1.7		1.5		2.8			

Software Dev. Phase = 7.8M/M 1296 hrs (1979 Dollars)
1296 hrs (overhead + G&A, Less Profit) = $38,880

Maintenance Phase 24 Calendar Months
100 hrs per 10,000 source instructions for 24 months
10 hrs. x 24 months = 1.5M/M = $ 7,200

$46,080

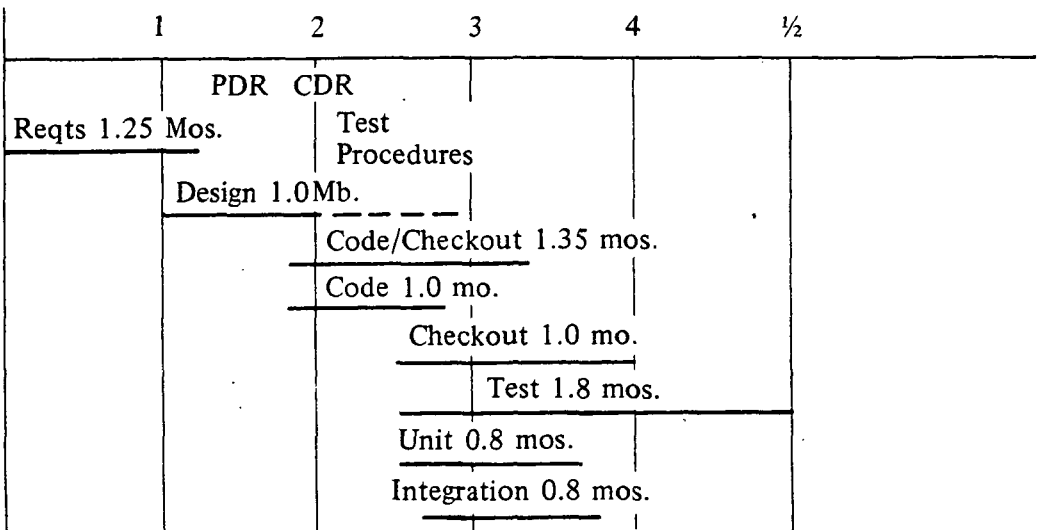
Software Dev. Computer Costs

- S/W will be developed on a dedicated Project Computer no additional costs. (Hardware costs to buy dedicated computer, not part of this estimate.) CPU hrs to be utilized 10 hrs.
- No additional Travel Costs for this Software effort.

W. Cheadle
Martin Marietta Aerospace

Budget for Software Development Phase				Calendar time for Software Development Phase
Reqts	23	1.8	3.5	4.5 Calendar months
Design	22	1.7		Arrived at by using our S/W scheduling model
Code	19	1.5	4.3	
Test	36	2.8		
	100%	7.8M/M		

HISTORY FILE UPDATE SCHEDULE — 4.5 Calendar Months



Man Loading						
Sys Reqts	0.6	0.1				0.7
Sys Design	0.4					0.4
S/W Reqts	0.6	0.1				0.7
Prel Design		0.5				0.5
Detail Design		1.2				1.2
Code		0.1	0.9			1.0
Checkout			0.4	0.1		0.5
Unit Test			0.4	0.2		0.6
Integration			0.1	0.5		0.6
System				0.8	0.8	1.6
	1.6	2.0	1.8	1.6	0.8	7.8M&M

W. Cheadle
Martin Marietta Aerospace

EARNED VALUE SYSTEM

Effort Required During First Month

• Software Development Plan update	0.2
• System Concept A Spec	0.3
• Functional Reqts B spec	0.7
• Interface control doc	0.1
• Prel users manual	0.1
• Sys Reqts review	0.1
• Sys design review	0.1
	1.6

2nd Month

• CPCI Code to spec	1.0
• Unit dev. folder	0.1
• CPCI test plan	0.4
• Reqts Traceability matrix	0.1
• PDR Support	0.2
• CDR Support	0.2
	2.0

3rd Month

• CPCI Test procedures	0.2
• Code effort	0.6
• Checkout	0.3
• User Manual	0.1
• Unit Dev. Folders	0.1
• Partial as built spec	0.1
• Test readiness review	0.1
• Run qualification tests	0.3
	1.8

4th Month

• Complete as built spec	0.1
• Version description doc.	0.1
• Test reports	0.3
• Qualification Tests	0.5
• Systems Test	0.6
	1.6

5th Month

• Sys test	0.5
• Test reports	0.3
	0.8

W. Cheadle
Martin Marietta Aerospace
7 of 7

L. Putnam
QSM

Advances have been made in the process of modeling the software cost and resource estimation process. One model that has been developed by QSM provides the user with estimates of software cost as well as the trade-offs that would be encountered by either shortening or extending the estimated development time. Pertinent and informative statistics are also provided in this model.

D3

SOFTWARE COSTING AND LIFE CYCLE CONTROL

©Lawrence H. Putnam
Quantitative Software Management, Inc.
1057 Waverley Way
McLean, VA 22101

It is remarkable that our $40–50 billion per year computer industry has 1/3 to 1/2 of its effort (and cost) out of control. I am referring to the software generation part of the industry. For 25 years now 200 to 300% cost overruns and up to 100% time slippages have been common, frequent—almost universal—as if there were no pattern, no process, no methodology, no characteristic behavior to the software development process. Indeed, it has become so unfathomable that responsible managers, controllers and corporate officers have tended to avoid the issue, accept the inevitability of overrun, and eat the extra cost—rather than find ways to get the problem solved.

If this were a trivial expense then such managerial responses would make sense. But $16–20 billion a year for the nation is non-trivial. Software development activities for major corporations cost 1–3% of revenues. This is perhaps 10–40% of net profit—thus, an activity worthy of controlling to the same standard as other critical corporate activities.

How can it be? People are aware of these realities. Many seminars, conferences and studies have been (and are still being) conducted to try to provide answers to the management questions:

Can I do it?
How much will it cost?
How long?
How many people?
What's the manloading?
What's the cash flow?
What's the trade-off?
What are the risks?

My studies of the past five years show very conclusively that there is a fundamental characteristic behavior to the software development process. The underlying characteristic is the complex human intercommunication process necessary to permit broad, abstract concepts to be transformed into a set of absolutely specific instructions the machine can respond to. This human intercommunication process is characterized by ambiguity and partial understanding. Progress proceeds in "fits and starts"—"surges"—"two steps forward, one back"—"loop back and start over," etc. These are all expressions to describe complex feedback paths, driven by random interaction among the human participants—all of whom must interact in a highly interdependent way.

People trying to plan and manage software attempt to do it deterministically – linear process flow diagram, decompose into a work breakdown structure and Gantt chart, assign tasks and schedule and then try to execute. Further, in an effort to meet arbitrary schedules, many activities that have sequential or partially sequential dependencies are attempted in parallel in the mistaken belief that what sometimes works in independent manufacturing processes will succeed in software. After 25 years of failure it is time to recognize this approach (by itself) will not work with software. We will have to deal with fundamentals.

The characteristic (average) behavior of software development over time is well described by the Rayleigh equation, a specific form of the Weibull family of reliability functions. The Rayleigh equation appears frequently in random statistical processes – scattering phenomena, narrow band Gaussian processes, diffusion and transport phenomena, quantum mechanics – so it is very reasonable to expect its appearance in software development where we implicitly recognize the unpredictability of the process, yet seem afraid to say it is a statistical process driven by many complex interactions unknown in advance and therefore random. The Rayleigh equation describes the average behavior over time of software development because it is a good model of a large number of Gaussian variables whose phases are random, meaning that many pieces of work will not be "in phase," hence will not "add" constructively, but may indeed "subtract," requiring feedback, rework, and so on.

Fortunately, we don't have to model this behavior in detail. The Rayleigh equation represents the overall time-varying behavior very well. Moreover, the Rayleigh equation parameters yield the management parameters that directly answer the management questions. The Rayleigh/Norden overall manpower equation for large systems is

$$\dot{y} = (K/t_d^2) \cdot t \cdot \exp(-t^2/2t_d^2) \text{ people}$$

where

K is the life cycle effort in manyears, or manmonths,

t_d is the development time in years, or months,

t is elapsed time in years, or months, from the beginning of detailed logic design and coding, and

$\dot{y}$ is manpower in manyears/year, or manmonths/month, or just plain, countable people at any instant in time.

Multiplying this equation by the labor rate turns it into a cost function. Integrating (adding up the curve) over time yields cumulative effort and cost any at time – thus, development effort and cost is an easily extractable subset of the life cycle numbers.

The relationships among the Rayleigh parameters are highly complex. This probably explains why purely empirical approaches have not yielded satisfactory solutions until now. Recently we have found good, practical ways to relate the Rayleigh management parameters to valid system characteristics in ways that answer the management questions directly with numbers that are the best possible answers. These findings are so important they should be commented upon immediately because the economic implications are absolutely awesome. The main points are these:

- A good, accurate method to size a system early in functional development has been developed.
- A software equation relating the system size to the managerial parameters – manmonths of effort (K), development time (t_d), and the state of technology being applied to the development effort – has been developed.
- Empirical verification from hundreds of systems of all types and development environments that the basic Rayleigh/Norden time varying behavior is phenomenologically sound.

- Empirical verification from the 400 odd systems collected by RADC that the parametric software equation and constraint relations are sound and are sophisticated enough to cope with the enormous range these parameters exhibit. (This has been the problem with single and multiple regression approaches – the variance has been enormous – this has been attributed to "poor data" when in reality, it is much more a function of the development environment (development computer, tools, and techniques) and system type (complexity).)

- Two good ways have been found to determine the management parameters from the system size and a set of system and managerial constraints.

  (1) LINEAR PROGRAMMING which produces a pair of constrained optimal solutions for the managerial parameters.

  (2) MONTE CARLO SIMULATION which produces a minimum time solution and uncertainty or risk profiles.

- Better and more straightforward ways to demonstrate the acute time sensitivity of the software development process.

- A dynamic (time varying) approach to measuring progress (not just resource consumption), coping with requirements changes when they occur and continually converging toward the actual system behavior – in effect, a real time process controller.

- The ability to play managerial "what if" games with software development projects AT ANY POINT IN THE LIFE CYCLE (from earliest feasibility analysis through development into the operations and maintenance phase).

I will comment on each of these points.

SIZING

Many software developers will tell you they cannot size a system accurately, that there is too much inherent uncertainty. This is partly true. They usually cannot size a module emanating from a functional description very accurately. But they can estimate ranges quite well. This is good enough because the statistics of aggregation work with us. We use the PERT estimating algorithm (Beta distribution) and ask our design engineers to estimate the size of each functional module in this way:

a – smallest number of source statements
m – most likely number of source statements
b – largest number of source statements

The expected number from a specific functional module is

$$\frac{a + 4m + b}{6}$$

and the associated standard deviation is approximately

$$\frac{|b - a|}{6}$$

When we aggregate all the module estimates into a systems estimate, a remarkable thing happens – the relative uncertainty of the system size ($^{\sigma}TOT/^{E}TOT$) is generally much smaller than the uncertainty (σ_i/E_i) of any of the modules. This is because of the cancelling effect that will occur in execution. Some modules will be smaller than planned, others larger; the net effect is a much smaller aggregate standard deviation than one would intuitively expect. Consider the following set of data obtained from an experienced team of about 15 system designers about twelve weeks into the functional design of a contemporary information retrieval system. Here are their estimates.

	Smallest	Most Likely	Largest	Expected	Std Dev
Maintain	8675.	13375.	18625.	13467.	1658.
Search	5577.	8988.	13125.	9109.	1258.
Route	3160.	3892.	8800.	4588.	940.
Status	850.	1425.	2925.	1579.	346.
Browse	1875.	4052.	8250.	4389.	1063.
Print	1437.	2455.	6125.	2897.	781.
User Aids	6875.	10625.	16250.	10938.	1563.
Incoming Messages	5830.	8962.	17750.	9905.	1987.
System Monitor	9375.	14625.	28000.	15979.	3104.
System Management	6300.	13700.	36250.	16225.	4992.
Comm. Proc..	5875.	8975.	14625.	9400.	1458.
Total				98475.	7081.

Note that the expected number of source statements for the system is just the sum of the expected number for each functional module. The standard deviation for the system is the square root of the sum of squares of the module standard deviations. This is what accounts for the cancelling effect. Note that the ratio $^{\sigma}TOT/^{E}TOT = 7081/98475$ is only about 7 percent, yet the coefficient of variation of one function, SYS MGT, is 4992/16225 – 31 percent, and the absolute magnitude of the standard deviation for SYS MGT, 4992, is 70 percent the size of the standard deviation for the entire system. The upshot of this is that we can predict system size to engineering accuracy even when there is large uncertainty in individual functional modules. This is a proven technique used in 15 years of experience in PERT charting. Counterintuitive –YES; but it works. Another point of major importance is that the engineers asked to provide the estimates are comfortable with the procedure. They are not threatened by range estimates. With this technique they can always be right, rather than always wrong as with any single number estimate they might provide. The more uncertain they are the broader the range they estimate. This is intelligent hedging that is accounted for in a systematic way. The technique has been used five times within GE with excellent results. Engineers and managers all felt comfortable with the procedure and satisfied with the results.

The question frequently arises as to why we estimate source statements instead of executable machine language instructions. The answer is simple and practical. Today, programmers and

analysts can estimate source statements because this is what comes out of their mind and off the tip of their pencil. Few people have any intuitive feel for executable machine langugae statements; the measure does not relate to their thinking or creative process. Both source statements and executable machine language instructions are valid information measures in the Shannon sense – they are ultimately reduced to bits of information in the machine. It is just that today, with most people writing in a language several levels above the machine level, source statements are natural; machine language instructions are not.

THE SOFTWARE EQUATION

The software equation $SS = C_k K^{1/3} t_d^{4/3}$ relates the number of source statements (SS) to the managerial parameters K and t_d. K is the life cycle size in manyears of effort; t_d is the development time in years. These are the Rayleigh/Norden parameters of the overall manpower equation.

$$\dot{y} = K/t_d^2 \cdot t \cdot e^{-t^2/2t_d^2} \text{ people}$$

C_k is a technology constant. It measures any throughput constraints that impede the progress of programmer/analysts – a batch development environment on a production machine (low) versus on-line, interactive program development on a dedicated test-bed machine (high). C_k is quantized. We see this in the data repeatedly. C_k varies in a set sequence of allowable values (Fibonacci sequence). The software equation will not be derived here; an adequate description of that process is contained in IEEE Transactions on Software Engineering, Vol. SE-4, No. 4, July 1978.

The important point is that the software equation gives us the linkage between system size, technological tools, effort and schedule. Effort and time are coupled. You cannot change one without changing the other. And the change is dramatic! Rearrange the software equation and you have the trade-off law:

$$\text{Dev Effort} = 0.4 \left(\frac{SS}{C_k}\right)^3 \frac{1}{t_d^4}$$

Note that Dev Effort = 0.4K; i.e., the area under the Rayleigh curve to t_d.

This turns immediately into the software economics law by multiplying by the burdened labor rate, $/MY.

$$\text{Dev Cost} = 0.4 (\$/MY) \frac{SS^3}{C_k^3 t_d^4}$$

All of these parameters can be favorably influenced by management before a project starts. Since they are all power functions and C_k goes up in quantum jumps by a factor of 1.6, then cost improvements by factors of 2 to 10 or more are possible with intelligent planning and good investment sense. The economics of this trade-off law are almost too good to be true. It says take 3 or 4 months longer and cut your cost in half; or better, buy a dedicated test-bed computer, thereby

increasing C_k by 1.6. When you cube this you have cut your cost by a factor of 4. Eliminate 10 percent of the system frills and shrink the number of source statements to 0.90SS. This cuts the cost to 73 percent of the original value. The improvements just cited for a 2-year system that we stretch out to 2.25 years are:

$$\text{Dev Cost}_{\text{before}} = 0.4\,(\$/\text{MY})\,\frac{SS^3}{C_k^3\,2^4} = 0.0625\left(0.4(\$/\text{MY})\,\frac{SS^3}{C_k^3}\right)$$

$$\text{Dev Cost}_{\text{after}} = 0.4\,(\$/\text{MY})\,\frac{(0.9)^3\;SS^3}{(1.6)^3\;C_k^3\;(2.25)^4} = 0.00694\left(0.4\$/\text{MY}\,\frac{SS^3}{C_k^3}\right)$$

The improvement ratio is:

$$\frac{\text{Dev Cost}_{\text{after}}}{\text{Dev Cost}_{\text{before}}} = \frac{0.00694}{0.0625} = \frac{1}{9}$$

The trade-off law is a consequence of system signal-to-noise ratio and bandwidth limitation: when development time ($B - 1/t_d$) is shortened, the bandwidth increases and signal-to-noise ratio decreases (actually, noise increases in the form of more difficult human intercommuncation). With the trade-off indicated, small time decreases soon make a system impossible to do – regardless of how many people or dollars are hurled at it. This is Brooks' Law at play.

We cope with the bandwidth limitation in the form of some empirically observed constraints that relate to the system difficulty K/t_d^2, the initial solpe of our Rayleigh manpower curve. The best measure seems to be the difficulty gradient $|\nabla D| \tilde{} K/t_d^3$. For a certain class of system, (new, stand-alone, etc.), the magnitude of this gradient stays constant.

When we solve the software equation simultaneously with the gradient constraint, we obtain the minimum time that a given size system can be built along with its associated life cycle effort, K(MY); development effort, 0.4K(MY), and development cost ($), $/MY (0.4K). These are expected values, of course, because of the inherent noise in the process.

EMPIRICAL SUPPORT

How can we be sure the software equation and gradient relation work across a broad class of system types and development environments? Classically, we use a set of data to determine the functional behavior, formulate a theory to explain the behavior and then verify the postulated behavior with another independent set of data. In this case, we found the basic behavior from the Army Computer Systems Command data, broadened the range of applicability with the Felix-Walston data (IBM Systems Journal, Vol. 16, No. 1, 1977) and recently have been able to verify the software equation, and gradient relations against the largest collection of software data yet collected. This is the software data base collected by Richard Nelson at Rome Air Development Center. Data for more than 400 systems have been collected and partially analyzed. Of particular interest are the machine generated plots of development effort, development time and average

manpower versus system size in delivered source lines of code (SS). The dependent variables are the management parameters and relate directly to our Rayleigh parameters. Development effort is 0.4K, development time is t_d, and average manpower is $[0.4(K/t_d)]$. When one looks at the Rome data polts (see Figures 1, 2, and 3), one notices the vast range of the independent variable from a hundred or so lines of code (small program) to systems of several million lines. The range of the dependent variables is large also. A clear functional behavior with good correlation is evident, but the value of the functions are severely limited as predictors because of the very large standard deviations. Some attribute this to "poor data." I submit it is inherent in any non-homogeneous data collection spanning many years, different languages, different system types, different design philosophies, etc. The variability combined with the good overall functional character is just what I have observed elsewhere with an independent data set. What it says to me is, "Hey, you've got a parametric variation present or an eigenvalue solution here – no single functional relation can handle it."

When I superimposed the software equation and the gradient constraint relation on the Rome data, I found a remarkably good fit. The slopes of the effort, manpower and duration curves of the functions obtained from the software equation were virtually the same as those determined by the RADC computer. However, no single technology constant (C_k) was capable of spanning the entire Rome data set. Indeed, it took two sequences of six or seven technology constants (ranging from about 600 to 14,000) to do this. No rational range of manpower for one technology constant can span the data range. For example, a range of 1 to 1,000 people working on a project will take in less than 1/2 the data points. The effort data say the same thing. The conclusion is that the real solution has to be parameterized; or be a discrete set of eigenvalues. The software equation, gradient and manpower constraints we have arrived at do span this data set, can rationally explain it, and the functional behavior is virtually the same as the data average; i.e., the Rome data 'proves' the software equation and constraint relations in all practical engineering respects. See Figures 1–5.

SIMULATION AND LINEAR PROGRAMMING

Management answers for effort, schedule and cost can be obtained using two powerful techniques that are well established.

Recall that our PERT estimate of source statements had associated with it a standard deviation reflecting the uncertainty in this estimate (and the nature of the way the programs and modules will be built; i.e., each program could be written many different ways to accomplish the same thing functionally; each of these would have a similar but different information content (bit count). The gradient relations were determined empirically and also had a statistical uncertainty in their determination. Now, if we let these two parameters vary randomly in a simultaneous solution that we run several thousand times, we can generate not only the expected value solutions for K and t_d, but also estimates of the standard deviations (more correctly, standard errors of the estimates). This is extremely valuable, because heretofore we have been totally mired in uncertainty – very precise single-value answers of completely unknown validity. Now, when the track record has been 200–300 percent overruns in cost and 50–100 percent overruns in schedule, decision makers do not believe single-value answers. They want to know the risk – the probability profile – they have been stung too often. In an immature discipline like software development (and the economics of it) managers need the risk information – and are entitled to it.

Linear programming lets us introduce the managerial constraints into the problem. Indeed, we can solve the linear programming problem with only the system size and the managerial constraints of maximum cost, maximum time, maximum peak manpower and minimum peak manpower, since these are all functions of our Rayleigh/Norden parameters; however, we also include system constraints such as difficulty and the difficulty gradient to prevent managers from attempting the impossible. The linear programming solution is possible because we can linearize the relations between our variables by taking logarithms and can express all the relations in terms of the two Rayleigh managerial parameters K and t_d. A two dimensional linear programming problem can be done graphically. Since our relations are linear in logarithms, we do it on log paper. The solution is trivial, but the insight and understanding in being able to visualize the interrelationships is rather profound. The minimum cost solution is immediately evident, the minimum time solution is immediately evident – the duals, maximum time and maximum cost, are also present as they must be in a linear programming solution – and the feasible trade-off range is identified in between the extrema. In being able to invoke this powerful technique, we produce constrained optimal solutions – the best that can be done within the constraints, and all other feasible choices. A graphical linear programming solution along with a brief write-up is attached. It works equally as well in the computerized simplex form. A sample output corresponding to the graphical solution is attached. Ideally, both these approaches should be combined – then managers can interactively iterate optimal solutions graphically on the CRT – using their constraints – until they have the best size, time, cost, manpower combination to meet their needs. With the smart graphics terminals available today, this can be done at negligible cost and time. See Figures 6-8.

SCHEDULE SENSITIVITY

Schedule is the most critical problem in software development. Software development acts like a low pass filter with sharp cut-off characteristics (call it a Rayleigh filter). This means that if the development time is arbitrarily specified by managerial fiat, then there is a high chance the system bandwidth will not match the arbitrary time (bandwidth) specified by management. This means the filter characteristic of the system will shape the input manpower and work profile to match as best it can. Attempts to force the system faster just generate power (manpower) losses. This can all be shown with vector arguments, Fourier analysis, and simulation. All methods give the same results – software development is very time sensitive – development time specification is not the prerogative of management – it belongs to the system. Management must iterate constraints to get into and stay within the feasible (schedule) region. (Fortunately, the linear programming solution bounds this region in time, effort, manpower, and cost.) To get some idea of development time sensitivity, consider the following table generated by simulation showing time sensitivity as a system size for a typical government development environment (CK = 5168).

Size (Source Stmts)		Dev Time (Months)		Dev Effort (MM)		Within Normal Range RADC Data Base?			
Avg	σ	Avg	σ	Avg	σ	MM	Dur	Avg # People	Prod
15,000	1500	12.9	0.6	34.7	6	Y	Y	Y	Y
50,000	5000	21.6	1.1	376.5	60	Y	Y	Y	Y
100,000	10000	29.1	1.4	992.9	152	Y	Y	Y	Y
250,000	25000	43.1	2.1	3188.2	498	N(>)	Y	N(>)	N(<)
500,000	50000	57.9	2.8	7782.3	1204	N(>)	Y	N(>)	N(<)

C_k = 5168, ∇D = 14.7, σD = 2.3

This table tells us that the time window is very narrow. For example, a 15,000 source statement system has a standard error of 0.6 month. If management picks the time at one year (very natural to do) then the probability of successful completion is small. 12.0 - 12.9 = -0.9 month, the no. std errors = -0.9/0.6 = -1.5, and $p\{t_d \leq 12 \text{ months}\}$ = 7% – certainly not odds for the betting man. However, a slip of a few weeks is usually forgivable by managers and customers so we hear little about these cases – nevertheless, the time sensitivity is there, but the absolute magnitude is below most people's response threshold. At the other extreme, 500,000 source statements, it is very hard to guess 57.9 months and 7782 manmonths. More managers and decision-makers would pick 48 months rather than 60 months knowing there is a better chance of getting funding. Yet 48 months is impossible (< 1% probability). Furthermore, 3 standard deviations in time (about 8.5 months) is easy to lose on a 5-year project. There are many external factors that can cause that much delay (late delivery of computer; late start with fixed end date, etc.). The only acceptable solution to this management dilemma is to get realistic time estimates, and then bias them for risk. Managers and decision-makers have to give up guessing schedules if they expect to succeed. The process is too counterintuitive, too time sensitive, making the guessing odds unacceptable.

DYNAMICS AND CONTROL OF THE PROCESS

I have described some good ways to estimate software projects BEFORE THEY START. But software development is a dynamic process. Requirements change. Functional descriptions change. Statutes change. All these things impact an ongoing software project. The change process may be so great that it invalidates a superb earlier estimate of size, cost, and schedule. So regardless of how good our prior estimate is, we still need to know what it is now, based on currently available information. We need a real-time process controller. This is nothing strange to us. It has been done in space operations. For example, we wouldn't think of sending astronauts to the moon if we couldn't measure where they were, compare it to where they should be and then make course corrections. The same concept can be applied to software development. We have a time-varying model that describes the expected manpower trajectory. All we need to do is feed it with the real data in real time so it can update and converge to the true (or present best estimate of the) trajectory. If we feed it manpower data, then we measure and predict resource consumption; but more importantly, if we feed it code production rate, we can measure, update, and predict task accomplishment – rate and % of source code complete. This lets us compare consumption versus accomplishment to see if the rates and predicted times are in agreement – a very important control checkpoint heretofor unavailable. This technique lets us control the process based on the existing system dynamics and make revised estimates of where we are heading.

We can also model the requirements change process in real time just before it takes place. This means decision-makers can know how much the change will cost over the life cycle, and what its slippage consequences are. The tecnhique is to use the 2nd order Rayleigh differential equation, solve it numerically in discrete steps and perturb the driving term by an amount proportional to the % change in the system (% of modules impacted, say). This linear approximation is representative and valid for the noise levels we are working within.

We apply the perturbation at the time the change is to occur and then project ahead to study the new predicted behavior compared to the predicted behavior before the change. Very complex situations can be modeled in this way with excellent indications of the expected response. The managerial insight one gains from this procedure is considerable – the "What if" possibilities

abound – "What if I double my effort for a month?" – "What if I am constrained on computer time for 2 months?" – "What if 25% of the system is cancelled half-way through development?" and so on. Graphical presentation of these situations on the CRT lets them be iterated and solved on-line interactively.

With these specific application techniques applied as described, we have been able to quantitatively come to grips with and produce reasonable engineering answers to the software cost estimating and life cycle control problem. We see that it is a more complicated problem than we would like it to be; yet, when we treat it as the time-varying problem it is, we see the solution is not as difficult as some that have been solved before in other fields – (indeed, we are able to pick up and use the best of those solutions in a number of cases) – and the solution can be easily updated wherever one is in the life cycle. The data requirements are small and occur naturally as a consequence of other normal reporting and record keeping; accordingly, the cost of driving the estimating and control system is negligible. The economics of the software development process is startling. The indications are clear that (apparently innocent) management choices can be made that affect cost by multiples of 5 to 10. With that kind of variation on multimillion dollar projects, managers need to know the choices, sensitivities and influences they can bring to bear – and they need it in numbers – over the whole life cycle.

The managerial questions – "Can I do it? How much? How long? How many people? What's the risk? What's the trade-off? – can be answered with numbers.

FIGURE 1.

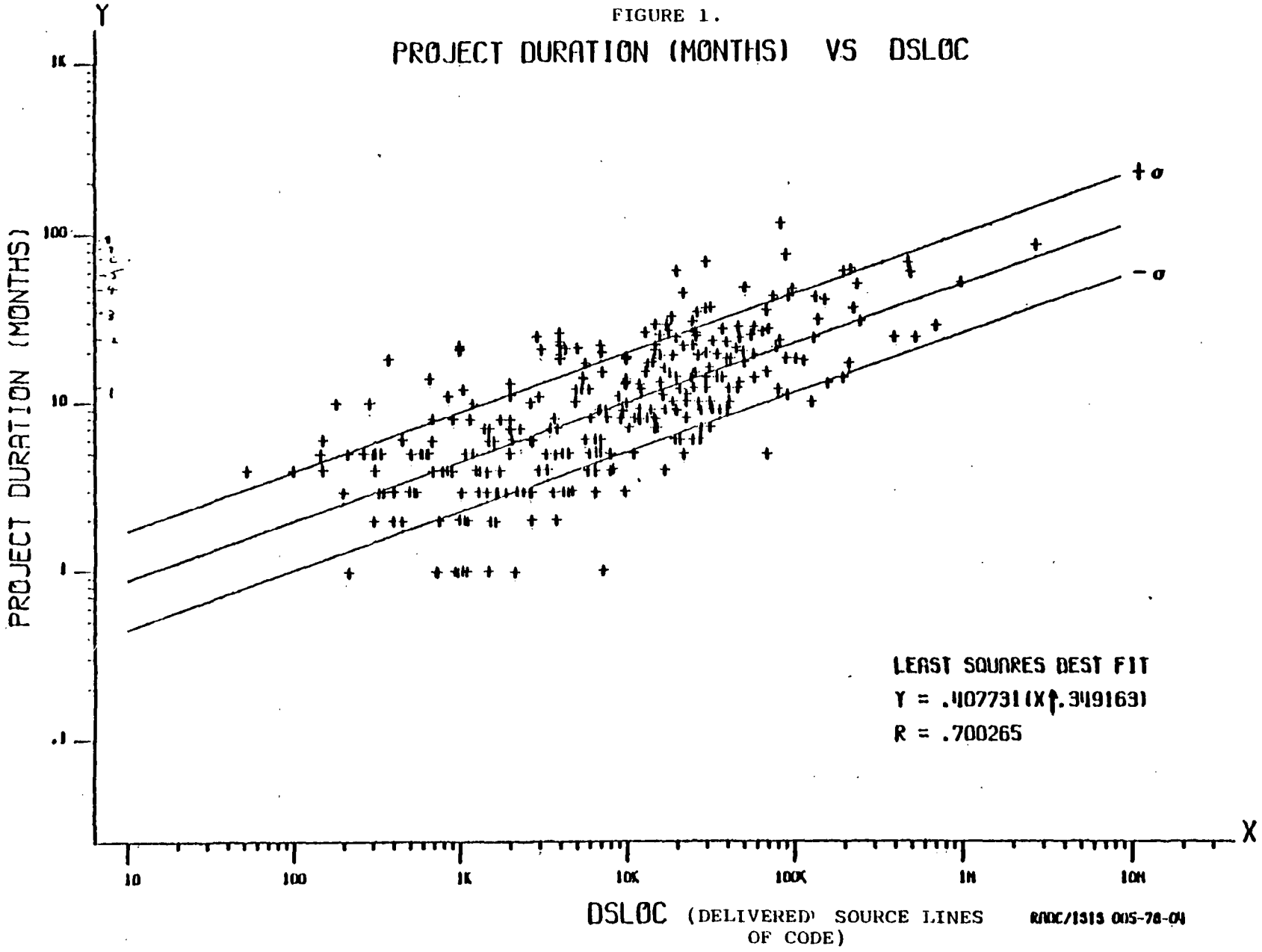

PROJECT DURATION (MONTHS) VS DSLOC
Y
X
PROJECT DURATION (MONTHS)
1K
100
10
1
.1
10
100
1K
10K
100K
1M
10M
+σ
−σ
LEAST SQUARES BEST FIT
Y = .407731(X↑.349163)
R = .700265
DSLOC (DELIVERED SOURCE LINES OF CODE)
RADC/1313 005-78-04

FIGURE 2.

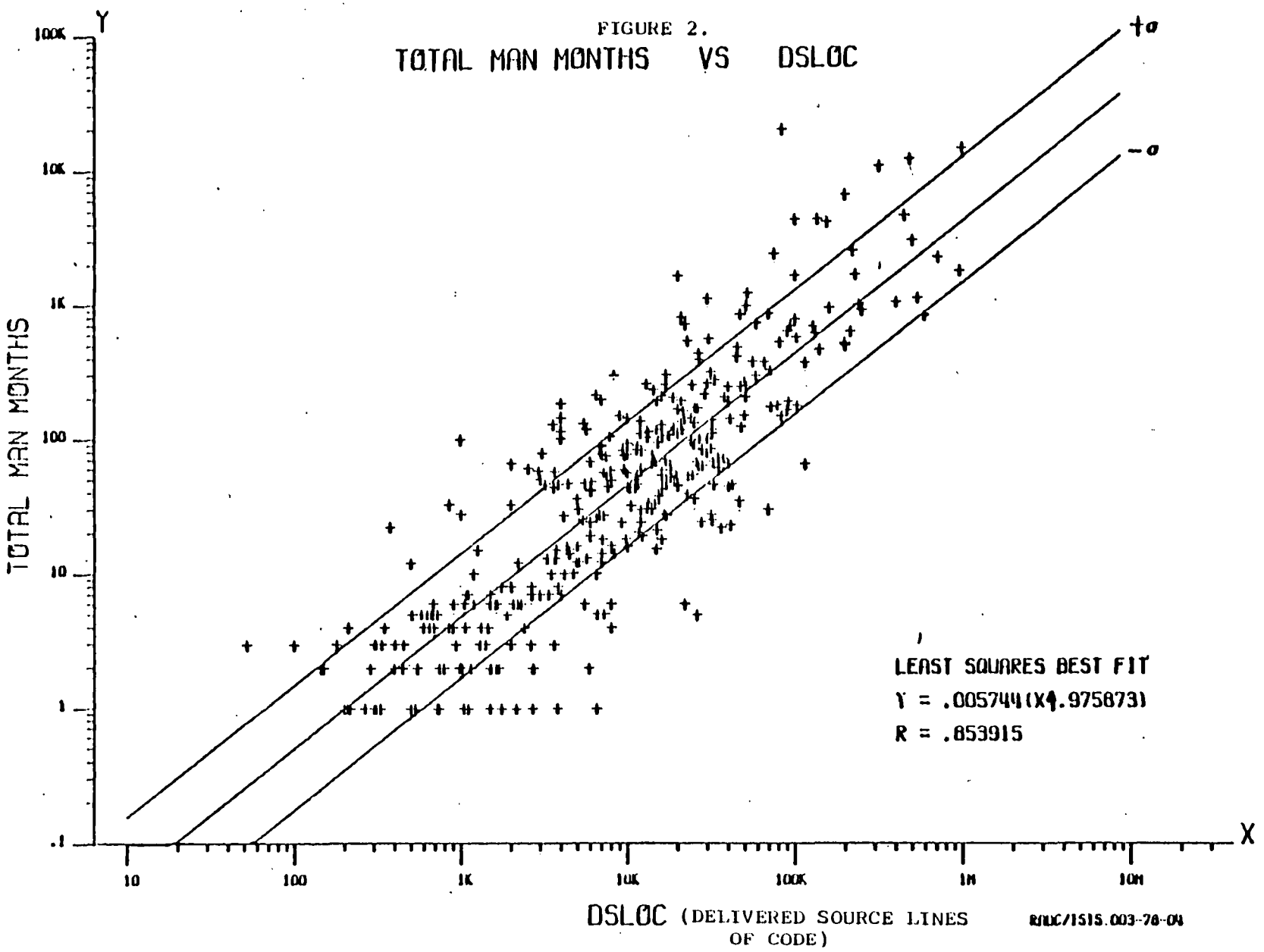

TOTAL MAN MONTHS VS DSLOC
Y
X
TOTAL MAN MONTHS
100K
10K
1K
100
10
1
.1
10
100
1K
10K
100K
1M
10M
+σ
−σ
LEAST SQUARES BEST FIT
Y = .005744(X↑.975873)
R = .853915
DSLOC (DELIVERED SOURCE LINES OF CODE)
RADC/1515.003-78-04

FIGURE 3.

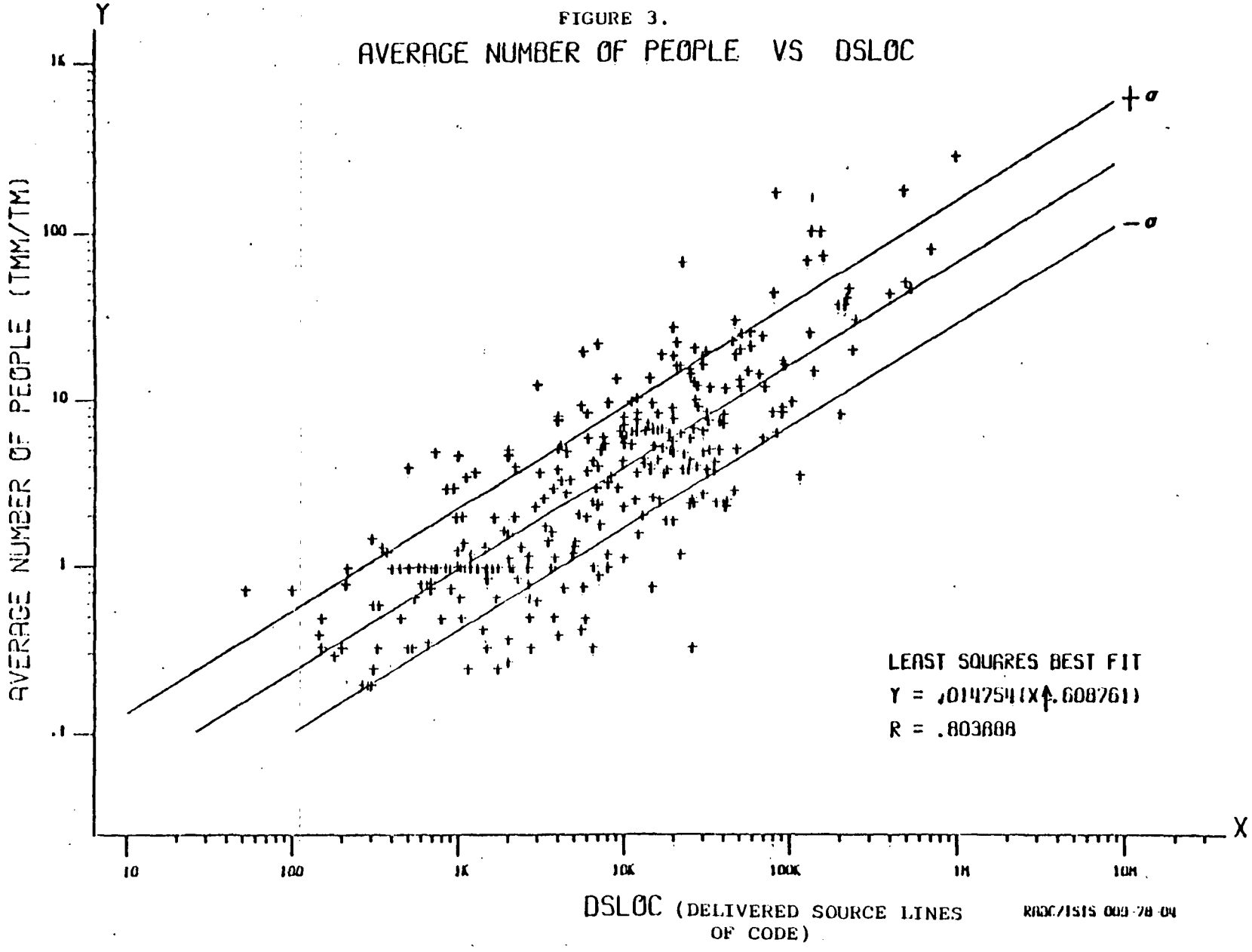

AVERAGE NUMBER OF PEOPLE VS DSLOC
Y
AVERAGE NUMBER OF PEOPLE (TMM/TM)
1K
100
10
1
.1
+σ
−σ
LEAST SQUARES BEST FIT
Y = .014754(X↑.608761)
R = .803888
X
10
100
1K
10K
100K
1M
10M
DSLOC (DELIVERED SOURCE LINES OF CODE)
RADC/ISIS 009-78-04

FIGURE 4.

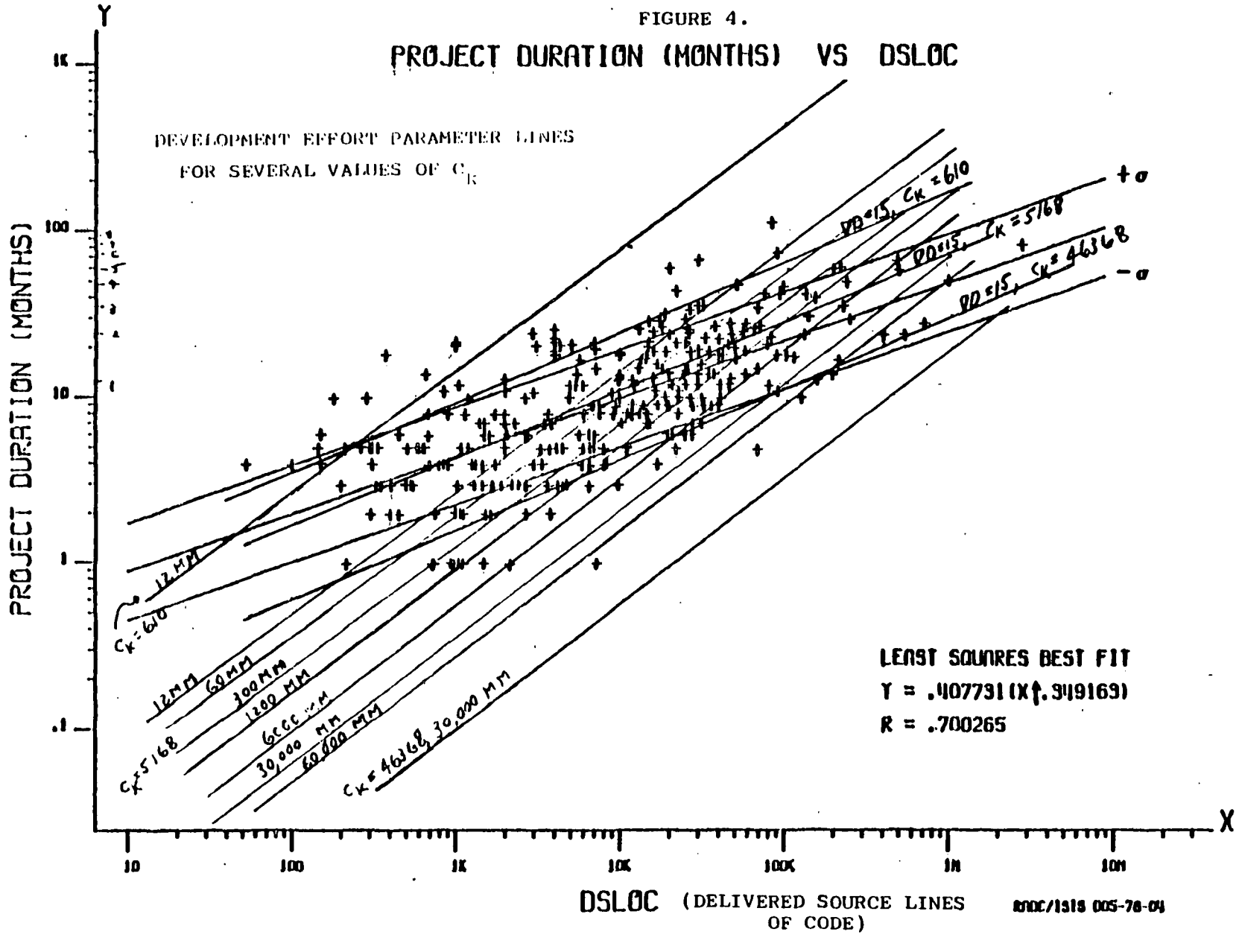

PROJECT DURATION (MONTHS) VS DSLOC
DEVELOPMENT EFFORT PARAMETER LINES
FOR SEVERAL VALUES OF C_K
PROJECT DURATION (MONTHS)
Y
X
1K
100
10
1
.1
10
100
1K
10K
100K
1M
10M
DSLOC (DELIVERED SOURCE LINES OF CODE)
+σ
-σ
PD=15, C_K=610
PD=15, C_K=5168
PD=15, C_K=46368
C_K=610
12 MM
60 MM
300 MM
1200 MM
6000 MM
30,000 MM
60,000 MM
C_K=5168
C_K=46368, 30,000 MM
LEAST SQUARES BEST FIT
Y = .407791 (X↑.349169)
R = .700265
BOOZ/1319 005-78-04

FIGURE 5.

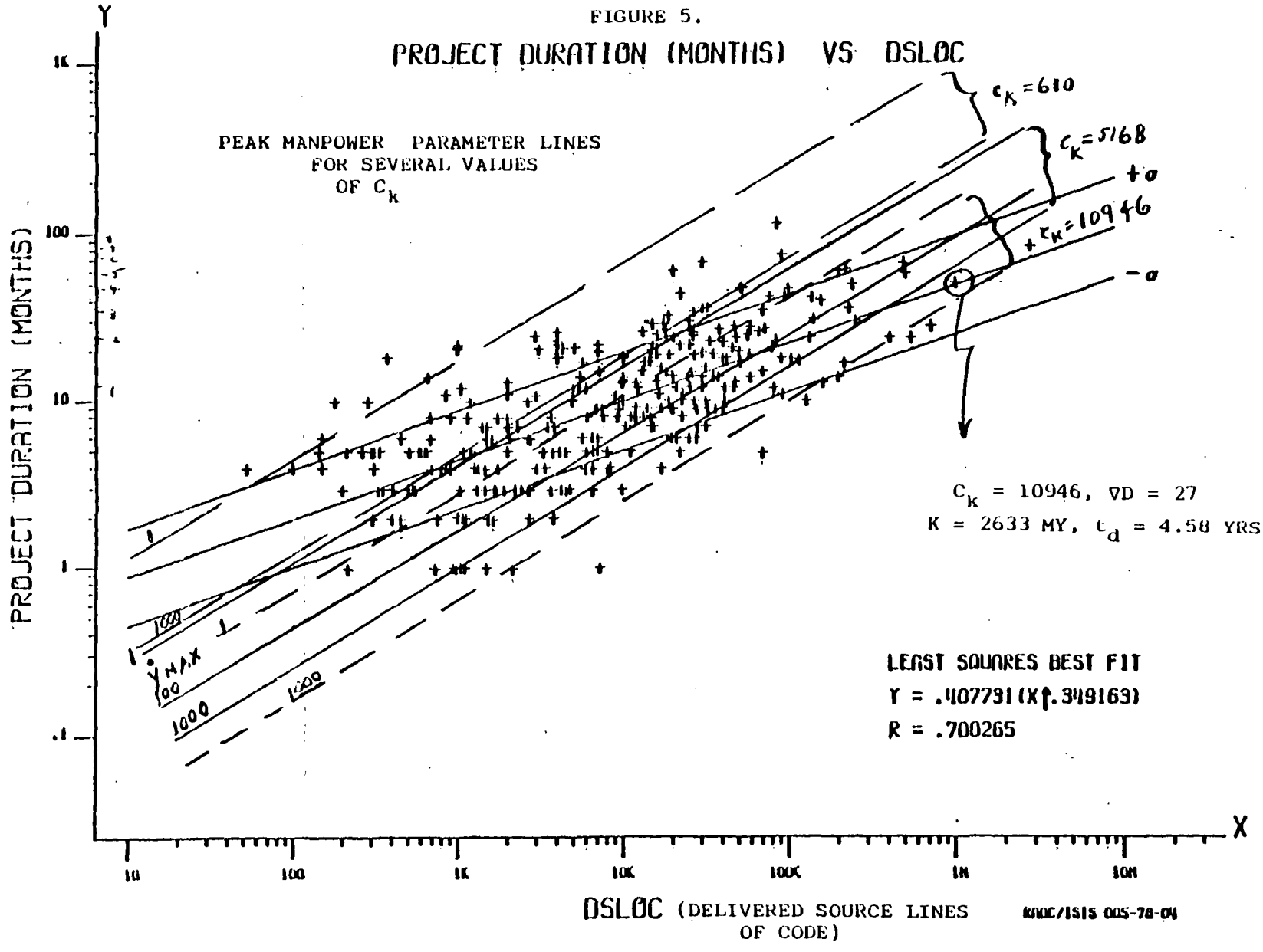

PROJECT DURATION (MONTHS) VS DSLOC
PEAK MANPOWER PARAMETER LINES FOR SEVERAL VALUES OF C_k
C_k = 610
C_k = 5168
+σ
C_k = 10946
−σ
C_k = 10946, VD = 27
K = 2633 MY, t_d = 4.58 YRS
LEAST SQUARES BEST FIT
Y = .407791 (X↑.349163)
R = .700265
PROJECT DURATION (MONTHS)
Y
X
1K
100
10
1
.1
10
100
1K
10K
100K
1M
10M
DSLOC (DELIVERED SOURCE LINES OF CODE)
RADC/ISIS 005-78-04

FIGURE 6.

Linear Programming Alternative

An alternative method for the Rayleigh parameter determination is linear programming. Since we are dealing with only two unknowns, K and t_d, and have a number of constraint conditions involving these parameters, we can easily turn it into a two dimensional linear programming problem which can be solved graphically. The nice feature of this approach is that a number of the constraints can be expressed directly in management terms. Design to cost and design to contract time is possible within the constrained optimization procedure. This procedure is outlined below. The following constraint conditions apply:

$S_s = C_K K^{1/3} t_d^{4/3}$	Software equation
$K/t_d \leq \sqrt{e}\ \dot{y}_{max}$	Maximum peak manpower
$K/t_d \geq \sqrt{e}\ \dot{y}_{max}$	Minimum peak manpower
$K/t_d^2 \leq \lvert D \rvert$	Maximum difficulty
$K/t_d^3 \leq \lvert \nabla D \rvert$	Maximum difficulty gradient
$t_d \leq$	contract delivery time
$\overline{S/MY}\ (.4K) \leq$	Total budgeted amount for development

These constraint conditions can be linearized by taking logarithms and using the simplex method of solving the linear programming problem. The simplest objective functions are cost and time. One generally wants to minimize one or the other of these. Typically we do both and then trade-off in the region in between.

Assume these constraints applied to SAVE:

Number of S_s = 98475

Maximum development cost $\leq$ $2 million

Maximum time (contract delivery) $\leq$ 2 years

Maximum manpower available at peak manning (hiring constraint, say) $\leq$ 28 people

Minimum manpower you desire to employ at peak manning $\geq$ 15 people

Maximum difficulty gradient $\leq$ 15

Maximum difficulty $\leq$ 50

Minimum productivity $\geq$ 2000 S_s/ MY

These translate into:

$$1/3 \log K + 4/3 \log t_d = \log 98475 - \log 10040$$

$$\log K = \log (2 \times 10^5/5 \times 10^4(.4))$$

$$\log t_d = \log 2$$

$$\log K - \log t_d = \log (\sqrt{e}\ 28)$$

$$\log K - \log t_d = \log (\sqrt{e}\ 15)$$

$$\log K - 3 \log t_d = \log 15$$

$$\log K - 2 \log t_d = \log 50$$

$$\log K = \log (98475/\ .4(2000))$$

The intersection of these lines bound the feasible region. An optimal solution will be at some intersection point. Further, because of the equality constraint it must be along the S_s = 98475 line. The limiting conditions in this case are: , $t_d \leq$ is 2 years, maximum peak manpower $\leq$ 28 people and S_s = 98475 source statements. Figure 4 shows the solution.

Reading off the solutions we see that:

	t_d (yrs)	K (MY)	E (MY)	$\overline{PR}$ (S_s/MY)
Minimum Time	1.83	84	33.6	98475/33.6 = 2931
Minimum Cost	2.0	61	24.4 $1.22M	98475/24.4 = 4036

Trade-off is possible along the S_s line between t_d = 1.83 years, K = 84 MY and t_d = 2 years, K = 61 MY without violating constraints.

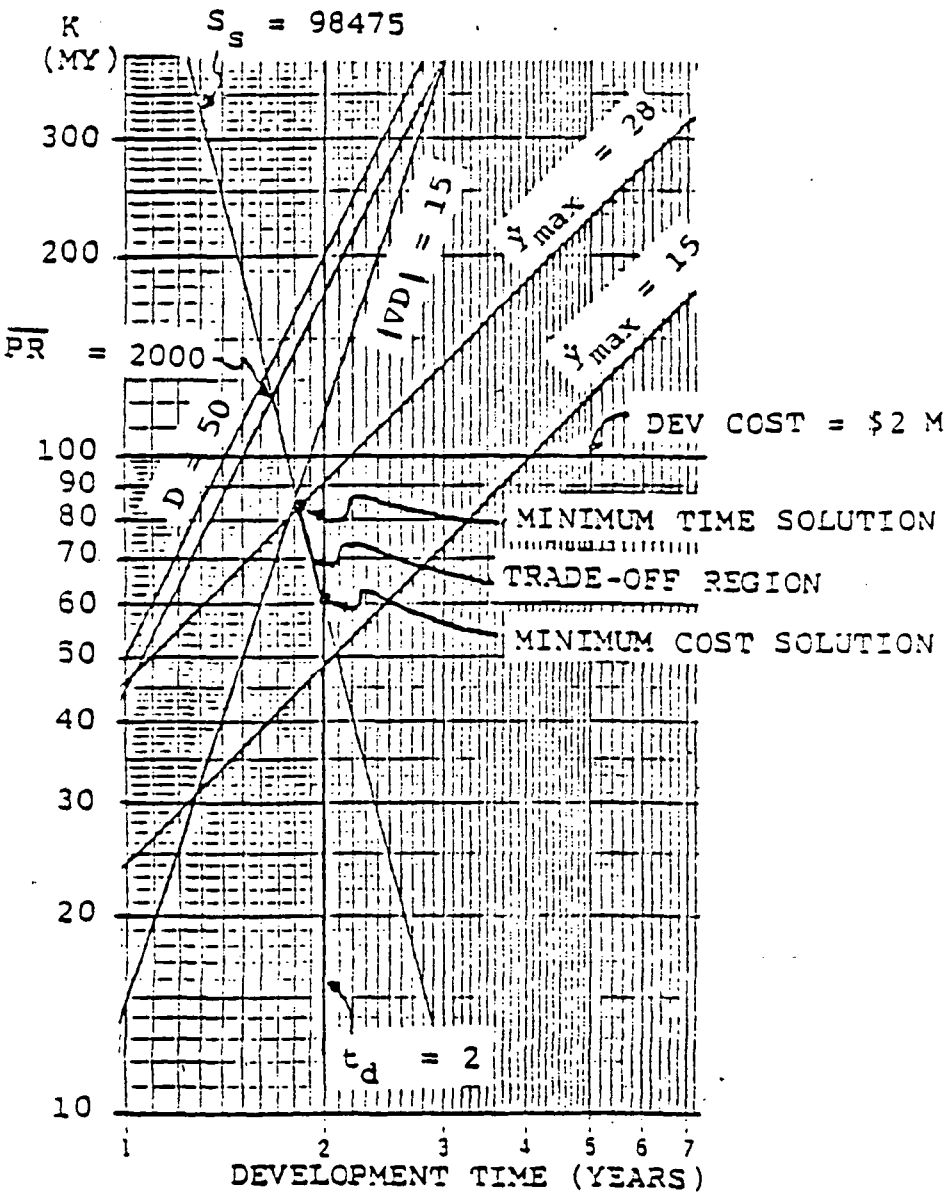
Here it is easy to see the counter intuitive nature of productivity. Note that productivity increases with development time because the required effort (E) goes down as time is increased.

One other point is important. If the technology constant is smaller, the S_s = 98475 line would shift parallel to the right (direction of increasing time). If the constraints remained numerically the same, the feasible region would change because of the relocation of the S_s line. The time constraint could probably not be met and a relaxation of that constraint would have to be sought.

This is a deterministic solution. However, by extending the idea of simulation, the linear programming concept can be embedded within a simulation and the uncertain constraints can be allowed to vary randomly about their mean values and the statistical uncertainty for the minimum time and minimum cost solutions can be obtained by running the problem a few thousand times.

FIGURE 7.

LINEAR PROGRAMMING SOLUTION
FOR SAVE



```
**********************************************************************
                          LINEAR PROGRAM                          20
**********************************************************************
TITLE:  SAVE                                        DATE: 17-Jan-79

   THIS FUNCTION USES THE TECHNIQUE OF LINEAR PROGRAMMING (SIMPLEX ALGORITHM)
   TO DETERMINE THE MINIMUM EFFORT (AND COST) OR THE MINIMUM TIME IN WHICH
   A SYSTEM CAN BE BUILT.  THE RESULTS ARE BASED ON THE ACTUAL MANPOWER, COST,
   AND SCHEDULE CONSTRAINTS OF THE USER, COMBINED WITH THE SYSTEM CONSTRAINTS
   YOU HAVE PROVIDED EARLIER TO YIELD A CONSTRAINED OPTIMAL SOLUTION.

 ENTER THE MAXIMUM DEVELOPMENT COST> 2000000

 ENTER MAXIMUM DEVELOPMENT TIME IN MONTHS> 24

 ENTER THE MINIMUM AND MAXIMUM NUMBER OF PEOPLE YOU
 CAN HAVE ON BOARD AT PEAK MANLOADING TIME> 15,28
```

	TIME	EFFORT	COST (X $1000)
MINIMUM COST	24.0 MONTHS	278. MM	1159.
MINIMUM TIME	21.9 MONTHS	399. MM	1662.

```
YOUR REALISTIC TRADE-OFF REGION LIES BETWEEN THE LIMITS OF THE TABLE ABOVE.

(INTERPOLATION IN THE TRADE-OFF TABLE BETWEEN THESE LIMITS WILL PRODUCE ALL
ACCEPTABLE ALTERNATIVES.  WOULD YOU LIKE TO SEE A TRADE-OFF ANALYSIS WITHIN
THESE LIMITS (Y OR N) ? Y
```

TIME	MANMONTHS	COST (X $1000)
21.93	399.	1662.
22.43	364.	1519.
22.93	334.	1390.
23.43	306.	1276.
24.00	278.	1159.

FIGURE 8.

VALIDATION OF THE SLIM METHODOLOGY TO ESTIMATING REAL TIME, COMMAND AND CONTROL APPLICATIONS DEVELOPMENT PROJECTS

Lawrence H. Putnam
Quantitative Software Management, Inc.
1057 Waverley Way
McLean, Virginia 22101

This set of visuals describes how SLIM (Software Life Cycle Management), an automated software cost estimating and life cycle planning tool belonging to Quantitative Software Management, Inc., was used to "replay" the development history of four real time, command and control system development projects done by Sperry Univac for the U.S. Air Force.

The development history (data) are taken from a Rome Air Development Center report and were incorporated into a thesis done at the Air Force Institute of Technology by a Captain Walker (AFIT/GCS/EE/78-21), who was working on variants of the Rayleigh/Norden Life Cycle Model used in SLIM. These data are shown in the next two pages as they appeared in Captain Walker's thesis. Manpower vs. time histories for 4 projects are given together with the more important aspects of the project and the development environment. This information is sufficient to calibrate SLIM, determine the technology constant representing complexity factors (like real time code) and environmental influences (tools, language, development discipline (MPP, TDSP, CPT, etc.)) and development constraints (development machine availability, batch vs on-line development, etc.) and then "replay" an idealization of the development time history as SLIM would have produced it.

This "replay" serves several useful purposes.

- It shows how easy it is to calibrate to past experience – thus tuning the estimating system to the skills, tools, and development, customer interface and administrative environment.
- It validates that the Rayleigh/Norden life cycle model (as implemented in SLIM) is a very satisfactory representation of what really happens in effectively managed software projects.
- It shows the model's adaptability to all size regimes of practical interest in the systems context (small – 16,000 HOL equivalent source statements example presented; medium – 46,000 HOL equivalent source statements example presented; and large – 500,000 HOL source statement example presented).
- It shows the specific applicability of the model to real time, command and control applications (Indeed, the model has been found to be applicable to any type of software system).
- It shows that the mixed language environment can be effectively handled by the SLIM methodology.

A few assumptions were made by me in fitting the data to the SLIM input file building editor. For example, the calendar starting dates were assumed since these were not given in the data. A

burdened labor rate of $50,000 per man year was assumed. An inflation rate of 6.5% was assumed for this time frame. All other relevant input information could be deduced from the development history obtained from the thesis. Only minor interpretation of this information was necessary.

Sperry Univac Programs 1 and 3 were done in a mix of languages. Sperry Univac Program 1 was 38% HOL and Sperry Univac Program 3 was 53% HOL. These were handled by converting to equivalent number of statements in one language or the other with due regard for the uncertainty in the conversion assumptions. Sperry Univac Program 1 was done both ways; converting everything to equivalent assembly language statements in the first case and converting everything to equivalent HOL statements in the second case. Very different technology constants were obtained; yet, because of the relationship exhibited by the software equation, $S_s = C_k K^{1/3} t_d^{4/3}$, nearly the same time-effort combination was obtained and a very similar time-varying manloading pattern emerged. In my opinion, the system acted more like an HOL development than an assembly language development and the fit seems to be slightly better.

The conversion process was handled this way for Sperry Univac Program 1. There were 90,000 DSLOC, 38%of which were HOL.

HOL Conversion

HOL Statements 0.38(90,000) = 34,200. We will assume an uncertainty on this of ±5000 HOL statements (Std Dev).

Assembly Statements 0.62(90,000) = 55,800. Assume possible conversion ratios from assembly to HOL:

	Equivalent HOL Statements
a (1% Prob.) 1 to 7	7971
m (most likely) 1 to 5	11160
b (99% Prob.) 1 to 3	18600

Using the PERT algorithm (modified)

a	m	b	Expected	Std Deviation
			34200	5000
7971	11160	18600	11868	2000 (1772 actual)
Expected HOL Equivalent Size			46068	
Approx. Standard Deviation on Size				5385 (RMS criterion)

The input to SLIM using the 99% range approach then is:

LOW: 46068 - 3(5385) = 29913 HOL Equivalent Statements

HIGH: 46068 + 3(5385) = 62223 HOL Equivalent Statements

with a normal disrtibution assumed.

The same procedure was used in converting the equivalent assembly language statements. The result obtained was an expected 226,800 equivalent assembly language instructions with an approximate standard deviation of 25,385 instructions.

Sperry Univac Program 3 was treated as an essentially HOL system (53% of the DSLOC) since a high percentage of the machine language instructions were HOL generated. This was born out by the manloading profile obtained from this conversion – characteristic of a small system with peak manpower obtained well prior to completion of development. An HOL to assembly conversion would have produced a profile with peak manpower occurring very close to the end of development – typical of large system behavior. The actual profile resembled the former rather than the latter confirming this reasoning.

A Sample Data

The data used in the sample calculations of Chapter IV was provided by Sperry-Univac Defense Systems in a Rome Air Development Center sponsored technical report (Ref. 23: 1-31). The data tabulated in Table A-I is the manning data for the four software systems reported in the report.

Table A-I I

Sperry-Univac Manning Data

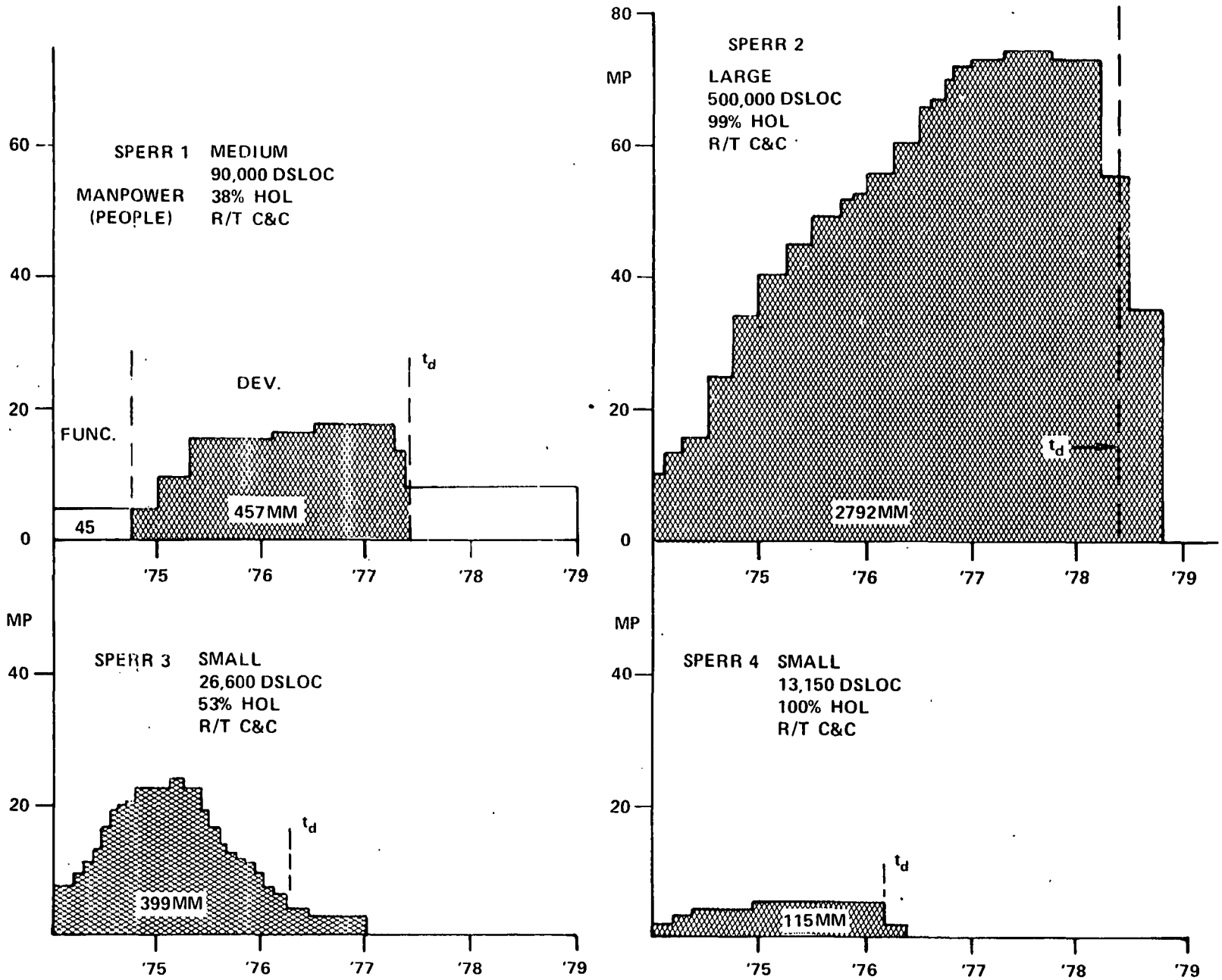
Month	Program				Month	Program				Month	Program		
	1	2	3	5		1	2	3	4		1	2	3
1	5	10	5	2	31	17	66	3	–	61	7	–	–
2	5	13	8	2	32	17	67	3	–	62	7	–	–
3	5	13	8	3	33	17	67	3	–	63	7	–	–
4	5	15	10	3	34	17	71	3	–	64	7	–	–
5	5	15	12	4	35	17	72	–	–	65	7	–	–
6	5	15	14	4	36	17	72	–	–	66	7	–	–
7	5	25	16	4	37	17	73	–	–	67	7	–	–
8	5	25	19	4	38	17	73	–	–	68	3	–	–
9	5	25	20	4	39	17	73	–	–	69	3	–	–
10	5	34	21	4	40	13	73	–	–	70	2	–	–
11	5	34	22	4	41	8	74	–	–	71	2	–	–
12	5	34	22	5	42	8	74	–	–	72	3	–	–
13	10	40	22	5	43	8	74	–	–	73	3	–	–
14	10	40	23	5	44	8	74	–	–	74	7	–	–
15	10	40	22	5	45	8	74	–	–	75	7	–	–
16	10	45	22	5	46	8	74	–	–	76	8	–	–
17	15	45	22	5	47	8	73	–	–	77	8	–	–
18	15	45	19	5	48	8	73	–	–	78	7	–	–
19	15	49	17	5	49	8	73	–	–	79	7	–	–
20	15	49	14	5	50	8	73	–	–	80	3	–	–
21	15	49	13	5	51	8	73	–	–	81	–	–	–
22	15	52	12	5	52	8	55	–	–				
23	15	53	11	5	53	8	55	–	–				
24	15	53	10	5	54	8	55	–	–				
25	15	56	8	5	55	8	35	–	–				
26	16	56	7	5	56	8	35	–	–				
27	16	56	6	1	57	8	35	–	–				
28	16	60	4	1	58	8	35	–	–				
29	16	60	4	–	59	8	–	–	–				
30	16	60	3	–	60	7	–	–	–				

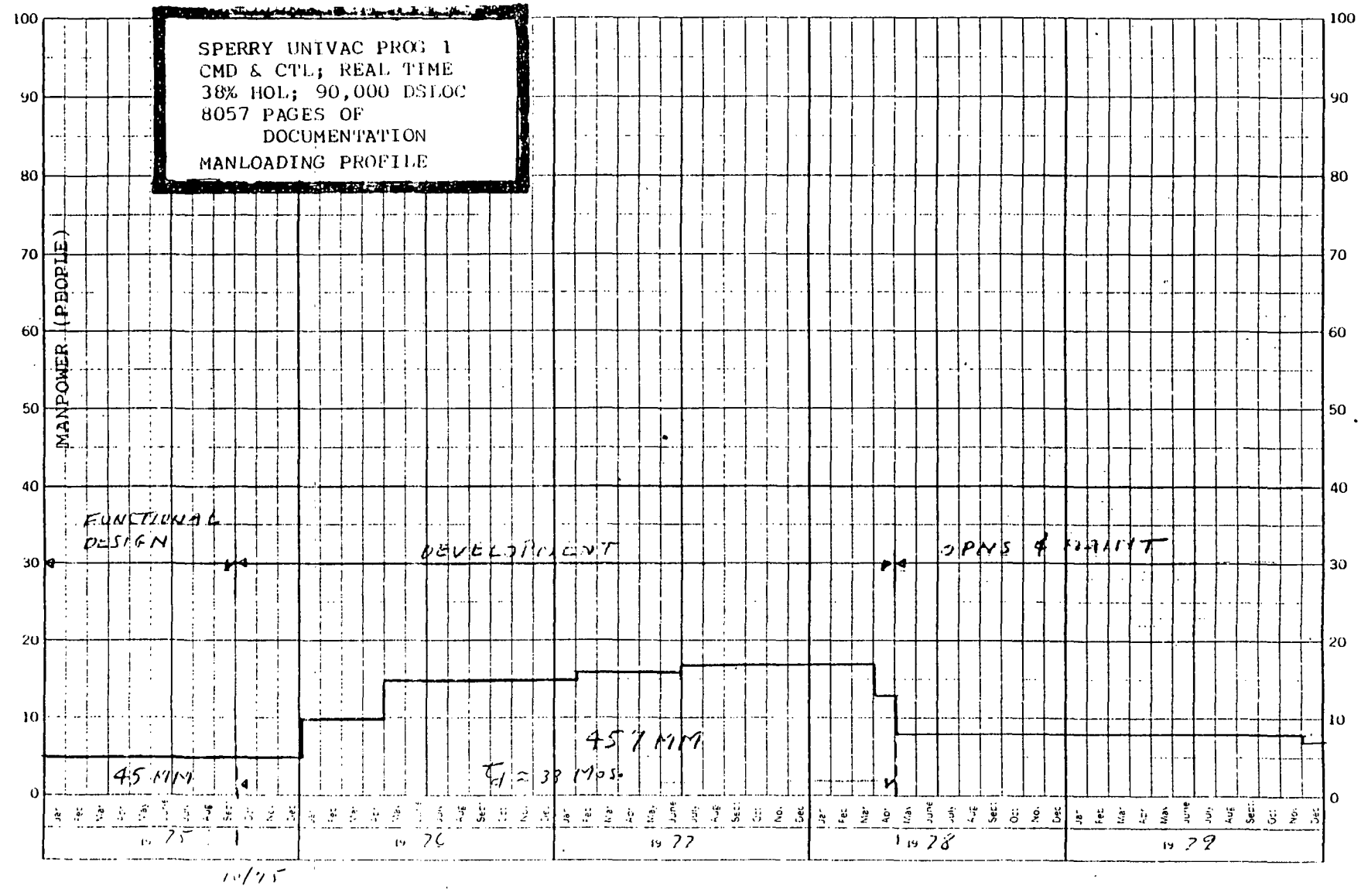
The units are man-months per month. Table A-II shows the factor data available on the four systems also found in the technical report.

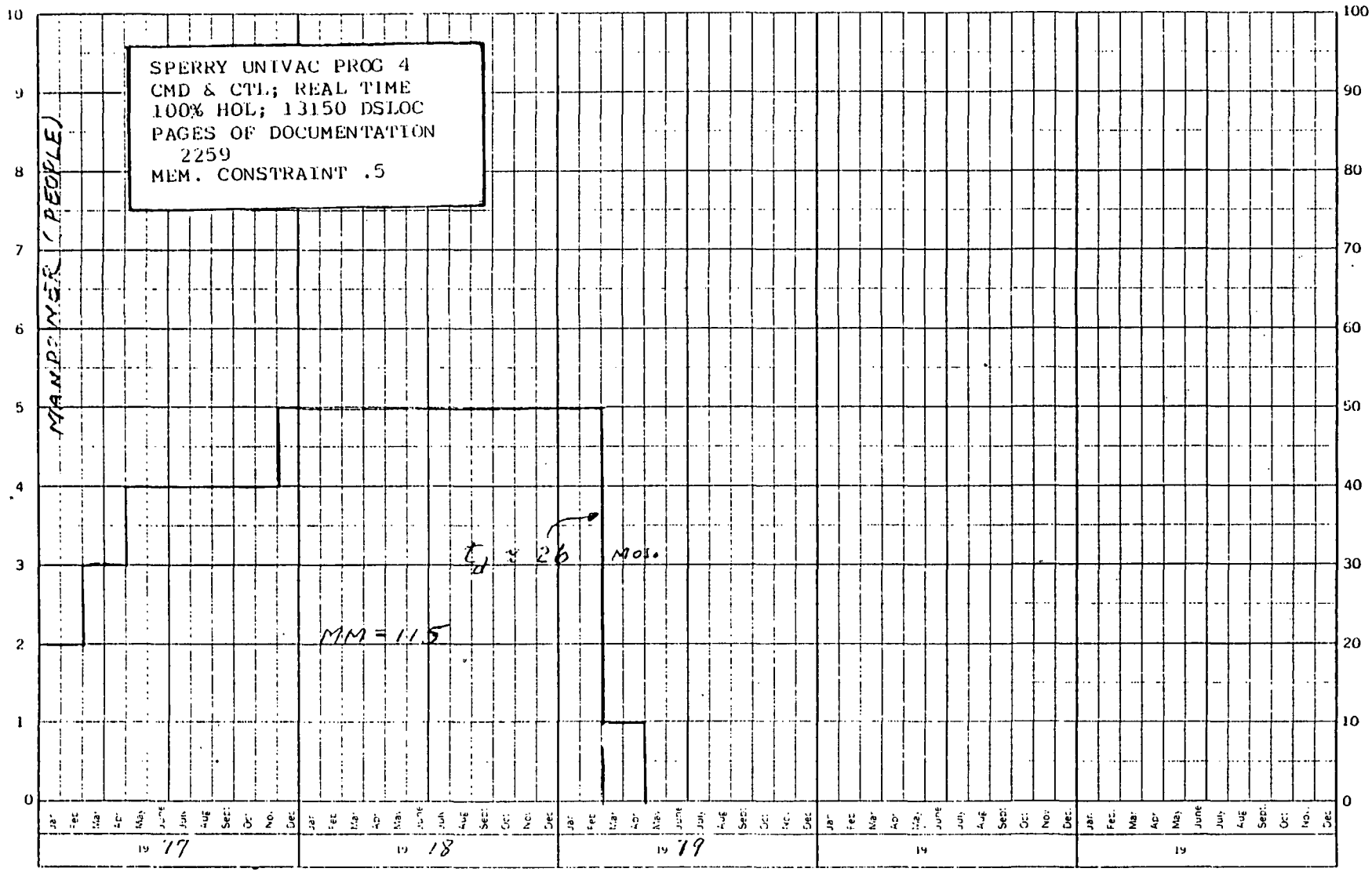
Table A-II

Sperry-Univac Factor Data

Factor	Program			
	1	2	3	4
Size in delivered source	90000	500000	26600	13150
Real-time application	1	1	1	1
Top-down structured design	0	0	1	1
Structured coding	1	0	0	1
Memory constraint	0.50	0.50	0.52	0.50
Percent HOL used	38	99	53	100
Programmer qualification education and training	39.0	37.1	62.8	82.4
Developed on target machine	1	1	0	0
Pages of documentation	8059	27014	3507	2259
Command and control application	1	1	1	1
Modular design	0	0	1	1
Program librarian	1	0	1	1
Structured narrative	1	0	0	1
Flow Charts	1	1	1	1


SPERR 1 MEDIUM
90,000 DSLOC
MANPOWER 38% HOL
(PEOPLE) R/T C&C
DEV.
FUNC.
45
457MM
t_d
SPERR 2
LARGE
500,000 DSLOC
99% HOL
R/T C&C
2792MM
SPERR 3 SMALL
26,600 DSLOC
53% HOL
R/T C&C
399MM
SPERR 4 SMALL
13,150 DSLOC
100% HOL
R/T C&C
115MM
MP
0
20
40
60
80
'75
'76
'77
'78
'79


SPERRY UNIVAC PROG 1
CMD & CTL; REAL TIME
38% HOL; 90,000 DSTOC
8057 PAGES OF
DOCUMENTATION
MANLOADING PROFILE
MANPOWER (PEOPLE)
FUNCTIONAL DESIGN
DEVELOPMENT
OPNS & MAINT
45 MM
45 MM
T_d = 38 Mos.
10/75
Jan Feb Mar Apr May June July Aug Sept Oct Nov Dec
'75
'76
'77
'78
'79



****

SUMMARY OF INPUT PARAMETERS

****

SYSTEM: SPERRY UNIVAC 4 DATE: 16-Nov-79

PROJECT START: 177

COST ELEMENTS

$/MY	50000.	INFLATION RATE	.080
STD DEV ($/MY)	5000.		

ENVIRONMENT

ONLINE DEV	0.00	HOL USAGE	1.00
DEVELOPMENT TIME	0.10	PRODUCTION TIME	0.90
LANGUAGE	JOVIAL		

SYSTEM

TYPE	COMMAND & CONTROL	REAL TIME CODE	0.30
LEVEL	1	UTILIZATION	0.50

MODERN PROGRAMMING PRACTICES

STRUCTURED PROG	3	DESIGN/CODE INSP	3
TOP-DOWN DEVELOPMENT	3	CHIEF PROGRAMMER TEAMS	3

EXPERIENCE

OVERALL	3	SYSTEM TYPE	2
LANGUAGE	3	HARDWARE	3

TECHNOLOGY

FACTOR	3	$C_K = 1220$	

SIZE

LOW	9150.	HIGH	17150.

****

****

SIMULATION

****

TITLE: SPERRY UNIVAC 4 DATE: 16-Nov-79

*** SIMULATION RUNNING - PLEASE WAIT ***

	MEAN	STD DEV
SYSTEM SIZE (STMTS)	13150.	1333.
MINIMUM DEVELOPMENT TIME (MONTHS)	25.0	1.2
DEVELOPMENT EFFORT (MANMONTHS)	125.8	20.1
DEVELOPMENT COST (X $1000) (UNINFLATED DOLLARS)	525.	98.
(INFLATED DOLLARS)	569.	107.

SENSITIVITY PROFILE FOR MINIMUM TIME SOLUTION
(EXPECTED VALUES OF TIME, EFFORT, AND COST FOR VARIOUS SYSTEM SIZES)

	SOURCE STMTS	MONTHS	MANMONTHS	COST (X $1000)
(-3 SD)	9150.	21.4	79.	329.
(-1 SD)	11817.	23.9	110.	457.
MOST LIKELY	13150.	25.0	126.	525.
(+1 SD)	14483.	26.1	142.	594.
(+3 SD)	17150.	28.0	177.	738.

A CONSISTENCY CHECK WITH DATA FROM OTHER SYSTEMS OF THE SAME SIZE SHOWS:

TOTAL MANMONTHS (126.)	WITHIN NORMAL RANGE
PROJECT DURATION (25.0 MONTHS)	LONGER THAN NORMAL TIME DURATION
AVG # PEOPLE(5.)	WITHIN NORMAL RANGE
PRODUCTIVITY (105. LINES/MM)	WITHIN NORMAL RANGE

****

****

MANLOADING

****

TITLE: SPERRY UNIVAC 4 DATE: 16-Nov-79

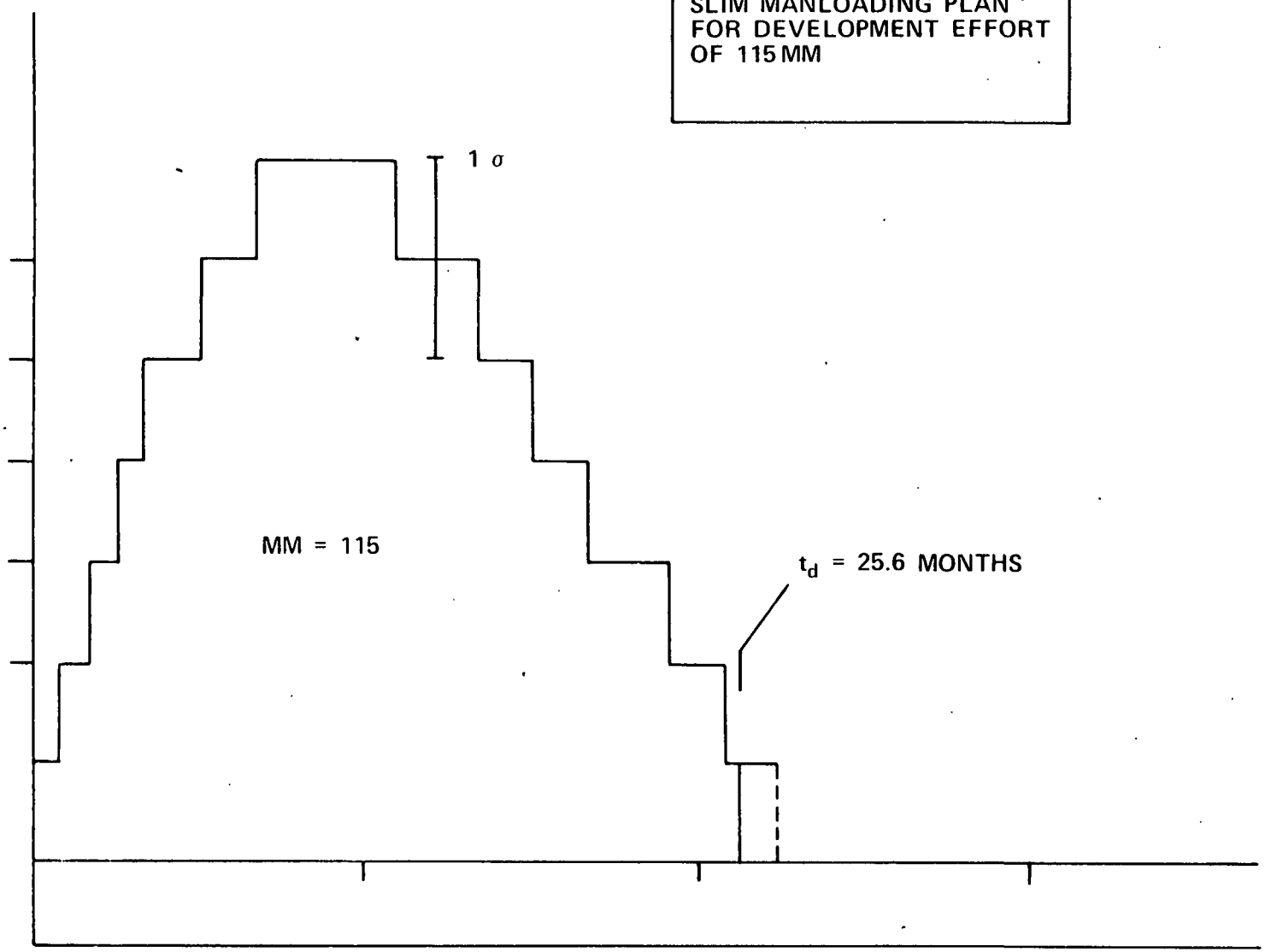
THE TABLE BELOW SHOWS THE MEAN PROJECTED EFFORT
AND ASSOCIATED (+ OR -) STANDARD DEVIATION REQUIRED
FOR DEVELOPMENT. THE INPUT PARAMETERS ARE:

	MEAN	STD DEV
DEVELOPMENT EFFORT (MM)	125.8	20.1
DEVELOPMENT TIME (MONTHS)	25.0	1.2

*** SIMULATION RUNNING - PLEASE WAIT ***

TIME	PEOPLE/MONTH	STD DEV	CUMULATIVE MANMONTHS	CUM STD DEV
JAN 77	1.	0.	1.	0.
FEB 77	2.	0.	3.	0.
MAR 77	3.	0.	6.	1.
APR 77	4.	1.	10.	2.
MAY 77	5.	1.	15.	2.
JUN 77	6.	1.	21.	3.
JUL 77	7.	1.	28.	5.
AUG 77	7.	1.	35.	6.
SEP 77	8.	1.	43.	7.
OCT 77	8.	1.	51.	8.
NOV 77	8.	1.	58.	9.
DEC 77	8.	1.	66.	11.
JAN 78	7.	1.	73.	12.
FEB 78	7.	1.	80.	13.
MAR 78	7.	1.	87.	14.
APR 78	6.	1.	93.	15.
MAY 78	6.	1.	99.	16.
JUN 78	5.	1.	104.	17.
JUL 78	5.	1.	108.	17.
AUG 78	4.	1.	112.	18.
SEP 78	3.	1.	116.	18.
OCT 78	3.	1.	119.	19.
NOV 78	2.	1.	121.	19.
DEC 78	2.	0.	123.	20.
JAN 79	2.	0.	125.	20.
FEB 79	1.	0.	126.	20.

SPERRY UNIVAC PROG 4
SLIM MANLOADING PLAN
FOR DEVELOPMENT EFFORT
OF 115 MM



****

LINEAR PROGRAM

****

TITLE: SPERRY UNIVAC 4 DATE: 16-Nov-79

THIS FUNCTION USES THE TECHNIQUE OF LINEAR PROGRAMMING (SIMPLEX ALGORITHM) TO DETERMINE THE MINIMUM EFFORT (AND COST) OR THE MINIMUM TIME IN WHICH A SYSTEM CAN BE BUILT. THE RESULTS ARE BASED ON THE ACTUAL MANPOWER, COST, AND SCHEDULE CONSTRAINTS OF THE USER, COMBINED WITH THE SYSTEM CONSTRAINTS YOU HAVE PROVIDED EARLIER TO YIELD A CONSTRAINED OPTIMAL SOLUTION.

ENTER THE MAXIMUM DEVELOPMENT COST IN DOLLARS> 1500000

ENTER MAXIMUM DEVELOPMENT TIME IN MONTHS> 28

ENTER THE MINIMUM AND MAXIMUM NUMBER OF PEOPLE YOU CAN HAVE ON BOARD AT PEAK MANLOADING TIME> 3,5

	TIME	EFFORT	COST (X $1000)
MINIMUM COST	28.0 MONTHS	80. MM	335.
MINIMUM TIME	27.4 MONTHS	88. MM	365.

MANPOWER CONSTRAINED

YOUR REALISTIC TRADE-OFF REGION LIES BETWEEN THE LIMITS OF THE TABLE ABOVE.

(INTERPOLATION IN THE TRADE-OFF TABLE BETWEEN THESE LIMITS WILL PRODUCE ALL ACCEPTABLE ALTERNATIVES. WOULD YOU LIKE TO SEE A TRADE-OFF ANALYSIS WITHIN THESE LIMITS (Y OR N) ? Y

TIME	MANMONTHS	COST (X $1000)
27.4	88.	365.
27.5	86.	360.
27.6	85.	355.
27.7	84.	349.
27.8	83.	344.
27.9	81.	340.
28.0	80.	335.

THE RESULTS SHOWN IN THIS TABLE CAN BE USED WITH DESIGN-TO-COST OR NEW TIME TO GENERATE AN UPDATED FILE AND AN ENTIRELY NEW ARRAY OF CONSEQUENT RESULTS FOR MANLOADING, CASHFLOW, LIFE CYCLE, RISK ANALYSIS, COMPUTER TIME AND FRONT END ESTIMATES.

****

****

NEW SCHEDULE DEFINITION

****

TITLE: SPERRY UNIVAC 4 DATE: 16-Nov-79

SLIM HAS PROVIDED ITS BEST ESTIMATE OF THE MINIMUM TIME AND CORRESPONDING EFFORT AND COST TO DEVELOP YOUR SYSTEM. THESE VALUES ARE:

MINIMUM TIME:		25.0 MONTHS
EFFORT:		126. MANMONTHS
COST (X $1000):	$	524.

A SHORTER DEVELOPMENT TIME CANNOT BE SPECIFIED ARBITRARILY BY THE USER. HOWEVER, IF A LONGER TIME (WITHIN REASONABLE LIMITS) IS SPECIFIED, THE SYSTEM CAN BE DEVELOPED FOR CONSIDERABLY LESS EFFORT - AND COST.

ENTER DESIRED DEVELOPMENT TIME IN MONTHS> 27.5

	MEAN	STD DEV
NEW DEVELOPMENT EFFORT (MANMONTHS)	86.	14.
NEW DEVELOPMENT COST (X $1000)	360.	57.

YOUR FILE IS UPDATED WITH THESE NEW PARAMETERS. RUN MANLOADING AND CASHFLOW OR LIFE CYCLE TO SEE HOW THESE SAVINGS CAN BE REALIZED.

A CONSISTENCY CHECK WITH DATA FROM OTHER SYSTEMS OF THE SAME SIZE SHOWS:

TOTAL MANMONTHS (86.)	WITHIN NORMAL RANGE
PROJECT DURATION (27.5 MONTHS)	LONGER THAN NORMAL TIME DURATION
AVG # PEOPLE(3.)	WITHIN NORMAL RANGE
PRODUCTIVITY (152. LINES/MM)	WITHIN NORMAL RANGE

****

****

SUMMARY OF INPUT PARAMETERS

****

TITLE: SPERRY UNIVAC 4 DATE: 16-Nov-79

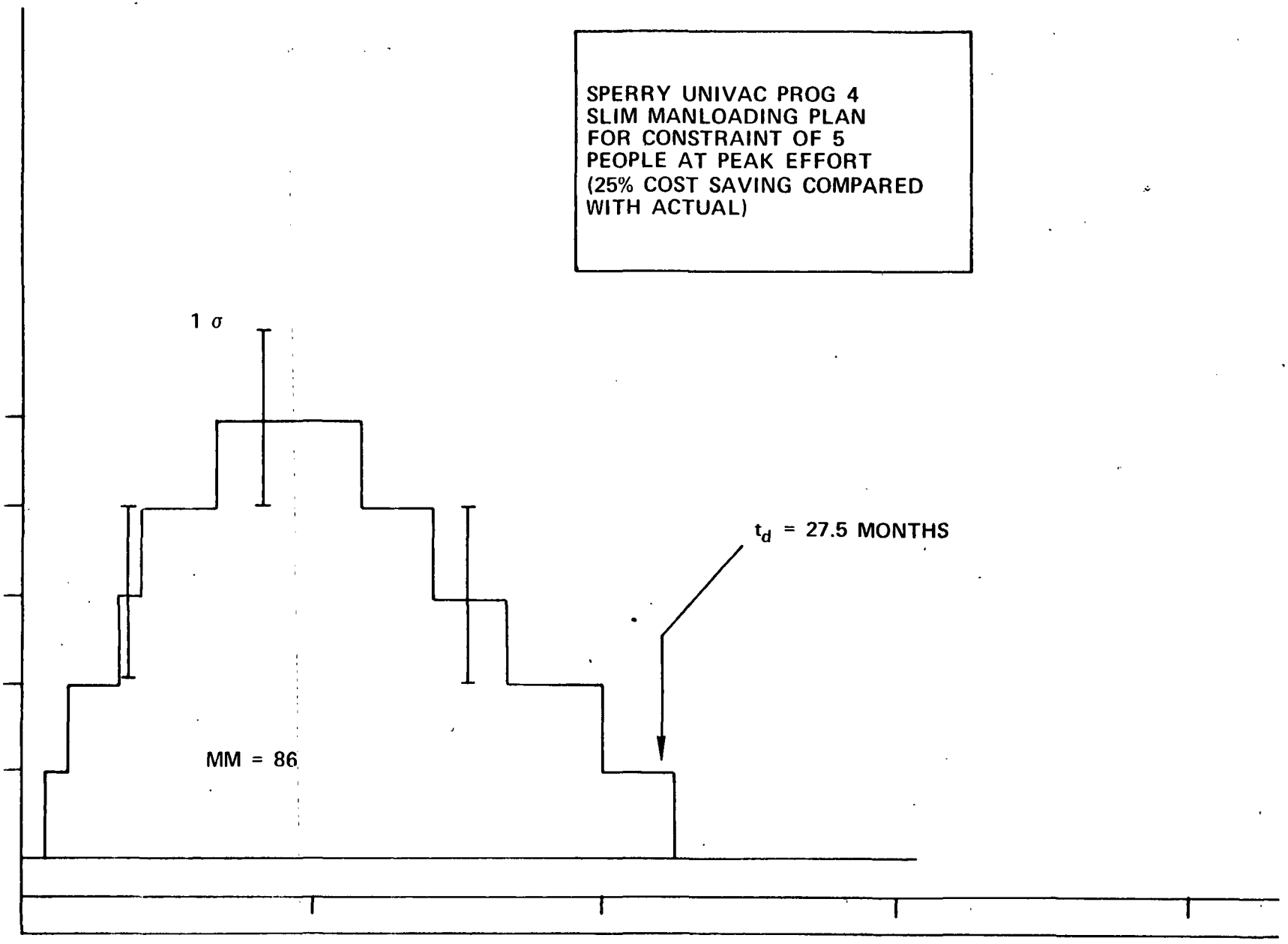
THE TABLE BELOW SHOWS THE MEAN PROJECTED EFFORT
AND ASSOCIATED (+ OR -) STANDARD DEVIATION REQUIRED
FOR DEVELOPMENT. THE INPUT PARAMETERS ARE:

	MEAN	STD DEV
DEVELOPMENT EFFORT (MM)	86.3	13.8
DEVELOPMENT TIME (MONTHS)	27.5	1.4

*** SIMULATION RUNNING - PLEASE WAIT ***

TIME	PEOPLE/MONTH		STD DEV	CUMULATIVE MANMONTHS	CUM STD DEV
JAN 77	0.		0.	0.	0.
FEB 77	1.		0.	1.	0.
MAR 77	2.		0.	3.	1.
APR 77	2.		0.	6.	1.
MAY 77	3.		1.	9.	1.
JUN 77	4.		1.	12.	2.
JUL 77	4.		1.	16.	3.
AUG 77	4.		1.	20.	3.
SEP 77	5.		1.	25.	4.
OCT 77	5.	PEAK	1.	30.	5.
NOV 77	5.		1.	35.	6.
DEC 77	5.		1.	39.	6.
JAN 78	5.	MP	1.	44.	7.
FEB 78	5.		1.	49.	8.
MAR 78	5.		1.	53.	9.
APR 78	4.		1.	58.	9.
MAY 78	4.		1.	62.	10.
JUN 78	4.		1.	65.	10.
JUL 78	3.		1.	69.	11.
AUG 78	3.		0.	72.	11.
SEP 78	3.		1.	75.	12.
OCT 78	2.		0.	77.	12.
NOV 78	2.		0.	79.	13.
DEC 78	2.		0.	81.	13.
JAN 79	2.		0.	83.	13.
FEB 79	1.		0.	84.	13.
MAR 79	1.		0.	85.	14.
APR 79	1.		0.	86.	14.
MAY 79	0.		0.	87.	14.

****


SPERRY UNIVAC PROG 4
SLIM MANLOADING PLAN
FOR CONSTRAINT OF 5
PEOPLE AT PEAK EFFORT
(25% COST SAVING COMPARED
WITH ACTUAL)
1 σ
t_d = 27.5 MONTHS
MM = 86

****

RISK ANALYSIS

****

TITLE: SPERRY UNIVAC 4 DATE: 16-Nov-79

THE TABLES BELOW SHOW THE PROBABILITY THAT IT WILL NOT TAKE MORE THAN THE INDICATED AMOUNT OF TIME, EFFORT, AND DOLLARS TO DEVELOP YOUR SYSTEM.

PROBABILITY	TIME (MONTHS)
1. %	24.3
5. %	25.3
10. %	25.8
20. %	26.4
30. %	26.8
40. %	27.2
EXPECTED → 50. %	27.5
60. %	27.8
70. %	28.2
80. %	28.6
90. %	29.2
95. %	29.7
99. %	30.7

PROBABILITY PROFILE

PROBABILITY	MANMONTHS	COST (X $1000)	INFLATED COST(X $1000)
1. %	54.	226.	247.
5. %	64.	265.	289.
10. %	69.	286.	312.
20. %	75.	311.	340.
30. %	79.	329.	360.
40. %	83.	345.	377.
→ 50. %	86.	360.	393.
60. %	90.	374.	409.
70. %	94.	390.	426.
80. %	98.	408.	445.
90. %	104.	433.	473.
95. %	109.	454.	496.
99. %	118.	493.	539.

PROBABILITY PROFILE

****

****

CALIBRATE

****

THIS FUNCTION ENABLES THE USER TO MAKE FUTURE ESTIMATES BASED ON HISTORICAL DATA FROM HIS ORGANIZATION AS WELL AS ON THE TYPE AND SIZE OF THE SYSTEM. IN ESSENCE, **CALIBRATE** TAKES TIME AND MANPOWER DATA FROM PAST SOFTWARE PROJECTS AND COMPUTES A TECHNOLOGY FACTOR FOR THE USER'S ORGANIZATION. THIS FACTOR IS REALLY AN INDICATION OF THE STATE OF TECHNOLOGY WHICH A PARTICULAR ORGANIZATION APPLIES TO A SOFTWARE PROJECT.

THE FOLLOWING HISTORICAL DATA IS REQUIRED:
(1) SYSTEM NAME (UP TO 20 CHARACTERS)
(2) TOTAL SYSTEM SIZE IN SOURCE STATEMENTS
(3) NUMBER OF MONTHS TO DEVELOP
(4) NUMBER OF MANMONTHS TO DEVELOP

HISTORICAL DATA WILL BE PROVIDED FOR HOW MANY SYSTEMS? 1

ENTER ALL DATA FOR EACH SYSTEM ON 1 LINE, SEPARATED BY COMMAS.

ENTER SYSTEM NAME, SIZE, MONTHS, AND MANMONTHS FOR SYSTEM 1.
> SPERR1,226800,33,357

SYSTEM NAME	SIZE	DEV. TIME (MONTHS)	DEV. EFFORT (MANMONTHS)	LEVEL	TECHNOLOGY FACTOR
SPERR1 ASSY.	226800.	33.0	357.0	1	13

NEW / INT.

$C_K = 13530$

AVERAGE TECHNOLOGY FACTOR IS 13.

****

SUMMARY OF INPUT PARAMETERS

****

SYSTEM: SPERRY UNIVAC PROG 1 DATE: 14-Nov-79

PROJECT START: 1075

COST ELEMENTS

$/MY	50000.	INFLATION RATE	.065
STD DEV ($/MY)	5000.		

ENVIRONMENT

ONLINE DEV	0.40	HOL USAGE	0.38
DEVELOPMENT TIME	1.00	PRODUCTION TIME	0.00
LANGUAGE	ASSEMBLER		

SYSTEM

TYPE	COMMAND & CONTROL	REAL TIME CODE	0.50
LEVEL	1 NEW W/ INT.	UTILIZATION	0.50

MODERN PROGRAMMING PRACTICES

STRUCTURED PROG	1	DESIGN/CODE INSP	2
TOP-DOWN DEVELOPMENT	2	CHIEF PROGRAMMER TEAMS	1

EXPERIENCE

OVERALL	2	SYSTEM TYPE	1
LANGUAGE	1	HARDWARE	2

TECHNOLOGY FACTOR 13 $C_K = 13530$

SIZE

LOW	150648	HIGH	302952

ASSY EQUIV.

****

****

SIMULATION

****

TITLE: SPERRY UNIVAC PROG 1 DATE: 14-Nov-79

*** SIMULATION RUNNING - PLEASE WAIT ***

ASSY. EQUIV.

	MEAN	STD DEV
SYSTEM SIZE (STMTS)	226800.	25384.
MINIMUM DEVELOPMENT TIME (MONTHS)	30.2	1.6
DEVELOPMENT EFFORT (MANMONTHS)	544.7	90.4
DEVELOPMENT COST (X $1000) (UNINFLATED DOLLARS)	2273.	441.
(INFLATED DOLLARS)	2461.	478.

SENSITIVITY PROFILE FOR MINIMUM TIME SOLUTION
(EXPECTED VALUES OF TIME, EFFORT, AND COST FOR VARIOUS SYSTEM SIZES)

	SOURCE STMTS	MONTHS	MANMONTHS	COST (X $1000)
(-3 SD)	150648.	25.4	326.	1358.
(-1 SD)	201416.	28.7	473.	1973.
MOST LIKELY	226800.	30.2	545.	2273.
(+1 SD)	252184.	31.6	632.	2634.
(+3 SD)	302952.	34.2	800.	3334.

A CONSISTENCY CHECK WITH DATA FROM OTHER SYSTEMS OF THE SAME SIZE SHOWS:

TOTAL MANMONTHS (545.)	WITHIN NORMAL RANGE
PROJECT DURATION (30.2 MONTHS)	WITHIN NORMAL RANGE
AVG # PEOPLE(18.)	WITHIN NORMAL RANGE
PRODUCTIVITY (416. LINES/MM)	WITHIN NORMAL RANGE

****

****

LINEAR PROGRAM

****

TITLE: SPERRY UNIVAC PROG 1 DATE: 14-Nov-79

THIS FUNCTION USES THE TECHNIQUE OF LINEAR PROGRAMMING (SIMPLEX ALGORITHM) TO DETERMINE THE MINIMUM EFFORT (AND COST) OR THE MINIMUM TIME IN WHICH A SYSTEM CAN BE BUILT. THE RESULTS ARE BASED ON THE ACTUAL MANPOWER, COST, AND SCHEDULE CONSTRAINTS OF THE USER, COMBINED WITH THE SYSTEM CONSTRAINTS YOU HAVE PROVIDED EARLIER TO YIELD A CONSTRAINED OPTIMAL SOLUTION.

ENTER THE MAXIMUM DEVELOPMENT COST IN DOLLARS> 2500000

ENTER MAXIMUM DEVELOPMENT TIME IN MONTHS> 36

ENTER THE MINIMUM AND MAXIMUM NUMBER OF PEOPLE YOU CAN HAVE ON BOARD AT PEAK MANLOADING TIME> 10,30

	TIME	EFFORT	COST (X $1000)
MINIMUM COST	36.0 MONTHS	275. MM	1144.
MINIMUM TIME	30.2 MONTHS	552. MM	2298.

YOUR REALISTIC TRADE-OFF REGION LIES BETWEEN THE LIMITS OF THE TABLE ABOVE.

(INTERPOLATION IN THE TRADE-OFF TABLE BETWEEN THESE LIMITS WILL PRODUCE ALL ACCEPTABLE ALTERNATIVES. WOULD YOU LIKE TO SEE A TRADE-OFF ANALYSIS WITHIN THESE LIMITS (Y OR N) ? Y

TIME	MANMONTHS	COST (X $1000)
30.2	552.	2298.
31.2	484.	2018.
32.2	427.	1779.
33.2	378.	1574.
34.2	336.	1398.
35.2	299.	1246.
36.0	275.	1144.

THE RESULTS SHOWN IN THIS TABLE CAN BE USED WITH DESIGN-TO-COST OR NEW TIME TO GENERATE AN UPDATED FILE AND AN ENTIRELY NEW ARRAY OF CONSEQUENT RESULTS FOR MANLOADING, CASHFLOW, LIFE CYCLE, RISK ANALYSIS, COMPUTER TIME AND FRONT END ESTIMATES.

****

****

FRONT-END ESTIMATES

****

TITLE: SPERRY UNIVAC PROG 1 DATE: 14-Nov-79

	TIME (MONTHS)			EFFORT (MM)		
	(LOW)	(EXPECTED)	(HIGH)	(LOW)	(EXPECTED)	(HIGH)
FEASIBILITY STUDY	6.3	7.5	8.8	8.	30.	53.
FUNCTIONAL DESIGN	8.4	10.1	11.7	54.	107.	161.

****

****

DESIGN TO COST

****

TITLE: SPERRY UNIVAC PROG 1 DATE: 14-Nov-79

SLIM HAS PROVIDED ITS BEST ESTIMATE OF THE MINIMUM TIME AND CORRESPONDING MAXIMUM EFFORT (AND COST) TO DEVELOP YOUR SYSTEM. THESE VALUES ARE:

MINIMUM TIME:		30.2 MONTHS
EFFORT:		545. MANMONTHS
COST (X $1000):	$	2270.

A GREATER EFFORT (OR COST) WOULD RESULT IN A VERY RISKY TIME SCHEDULE. HOWEVER, IF A LOWER EFFORT IS SPECIFIED (WITHIN REASONABLE LIMITS), DEVELOPMENT IS STILL FEASIBLE AS LONG AS YOU CAN TAKE MORE TIME.

ENTER DESIRED EFFORT IN MANMONTHS> 457

		MEAN	STD DEV
NEW DEVELOPMENT TIME (MONTHS)		31.7	1.7
NEW DEVELOPMENT COST (X $1000)	$	1904.	316.

YOUR FILE IS UPDATED WITH THESE NEW PARAMETERS. RUN MANLOADING AND CASHFLOW OR LIFE CYCLE TO SEE HOW THESE SAVINGS CAN BE REALIZED.

A CONSISTENCY CHECK WITH DATA FROM OTHER SYSTEMS OF THE SAME SIZE SHOWS:

TOTAL MANMONTHS (457.)	WITHIN NORMAL RANGE
PROJECT DURATION (31.7 MONTHS)	WITHIN NORMAL RANGE
AVG # PEOPLE(14.)	WITHIN NORMAL RANGE
PRODUCTIVITY (496. LINES/MM)	WITHIN NORMAL RANGE

****

****

LIFE CYCLE

****

SYSTEM: SPERRY UNIVAC PROG 1 DATE: 14-Nov-79

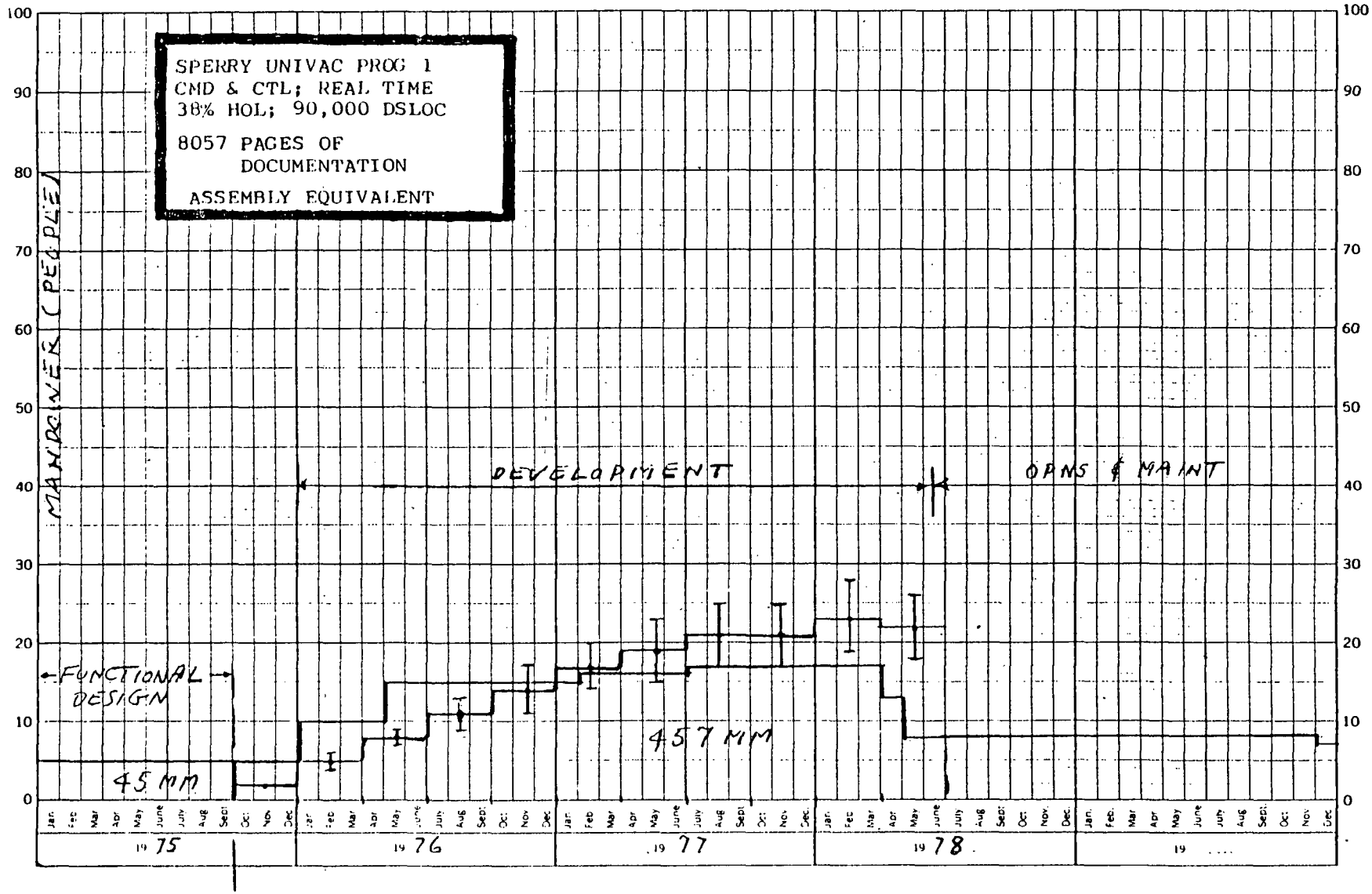
ASSEMBLY EQUIV.

THE TABLE BELOW SHOWS THE MEAN PROJECTED EFFORT AND CASHFLOW (AND ASSOCIATED STANDARD DEVIATIONS) OVER THE LIFE CYCLE OF THE SYSTEM. ALL PROJECTIONS ARE BASED ON AN OPTIMAL APPLICATION OF RESOURCES OVER TIME. THE INPUT PARAMETERS ARE:

	MEAN	STD DEV
DEVELOPMENT TIME (MONTHS)	31.7	1.7
LIFE CYCLE EFFORT(MM)	1161.5	192.7
AVG COST/MY (X $1000)	50.	5.
INFLATION RATE	0.065	0.010

QTR ENDING	PEOPLE		COST/QTR (X $1000)		CUM COST (X $1000)	
	MEAN	STD DEV	MEAN	STD DEV	MEAN	STD DEV
DEC 75	2.	0.	22.	5.	22.	4.
MAR 76	5.	1.	67.	17.	89.	17.
JUN 76	8.	1.	108.	21.	197.	38.
SEP 76	11.	2.	152.	36.	347.	68.
DEC 76	14.	3.	189.	46.	537.	104.
MAR 77	17.	3.	226.	47.	761.	148.
JUN 77	19.	4.	260.	62.	1019.	198.
SEP 77	21.	4.	291.	62.	1307.	254.
DEC 77	21.	4.	303.	64.	1614.	314.
MAR 78	23.	4.	331.	65.	1937.	376.
JUN 78	22.	4.	322.	65.	2267.	441.
SEP 78	22.	3.	333.	57.	2595.	504.
DEC 78	22.	4.	330.	62.	2924.	568.
MAR 79	21.	4.	321.	70.	3245.	631.
JUN 79	20.	3.	310.	63.	3554.	691.
SEP 79	18.	3.	282.	54.	3841.	747.
DEC 79	17.	3.	273.	55.	4110.	799.
MAR 80	15.	2.	256.	52.	4364.	848.
JUN 80	14.	3.	234.	53.	4598.	894.
SEP 80	12.	2.	210.	43.	4808.	935.
DEC 80	11.	2.	188.	43.	4996.	971.
MAR 81	9.	2.	162.	36.	5161.	1003.
JUN 81	8.	2.	142.	37.	5303.	1031.
SEP 81	7.	2.	127.	32.	5428.	1055.
DEC 81	6.	2.	112.	32.	5539.	1076.
MAR 82	5.	1.	93.	26.	5633.	1095.
JUN 82	4.	1.	77.	23.	5710.	1110.
SEP 82	3.	1.	63.	19.	5774.	1122.
DEC 82	3.	1.	50.	18.	5825.	1132.
MAR 83	2.	1.	45.	16.	5868.	1140.
JUN 82	2.	1.	34.	14.	5903.	1147.
SEP 82	1.	0.	27.	11.	5930.	1153.

LIFE CYCLE PROJECTIONS


SPERRY UNIVAC PROG 1
CMD & CTL; REAL TIME
38% HOL; 90,000 DSLOC
8057 PAGES OF
DOCUMENTATION
ASSEMBLY EQUIVALENT
MANPOWER (PEOPLE)
DEVELOPMENT
OPNS & MAINT
FUNCTIONAL
DESIGN
45 MM
457 MM
100
90
80
70
60
50
40
30
20
10
0
Jan Feb Mar Apr May June July Aug Sept Oct Nov Dec
19 75
19 76
19 77
19 78
19

****

RISK ANALYSIS

****

TITLE: SPERRY UNIVAC PROG 1 DATE: 14-Nov-79

THE TABLES BELOW SHOW THE PROBABILITY THAT IT WILL NOT TAKE MORE THAN THE INDICATED AMOUNT OF TIME, EFFORT, AND DOLLARS TO DEVELOP YOUR SYSTEM.

PROBABILITY	TIME (MONTHS)
1. %	27.8
5. %	28.9
10. %	29.6
20. %	30.3
30. %	30.8
40. %	31.3
EXPECTED → 50. %	31.7
60. %	32.1
70. %	32.6
80. %	33.1
90. %	33.8
95. %	34.4
99. %	35.6

PROBABILITY PROFILE

PROBABILITY	MANMONTHS	COST (X $1000)	INFLATED COST(X $1000)
1. %	281.	1169.	1271.
5. %	332.	1384.	1505.
10. %	360.	1499.	1629.
20. %	393.	1638.	1780.
30. %	417.	1739.	1889.
40. %	438.	1824.	1982.
→ 50. %	457.	1904.	2069.
60. %	476.	1984.	2156.
70. %	497.	2070.	2249.
80. %	521.	2170.	2358.
90. %	554.	2309.	2509.
95. %	582.	2424.	2634.
99. %	633.	2639.	2868.

PROBABILITY PROFILE

****

****

DOCUMENTATION

****

TITLE: SPERRY UNIVAC PROG 1 DATE: 14-Nov-79

IT IS POSSIBLE TO ESTIMATE THE NUMBER OF PAGES OF DOCUMENTATION, BASED ON DATA COLLECTED FROM SEVERAL HUNDRED SYSTEMS.

THE EXPECTED NUMBER FOR YOUR SYSTEM IS 15876 PAGES.

THE 90% RANGE IS FROM 4536 TO 38556 PAGES.

****

ACTUAL: 8057

```
ENTER SYSTEM NAME, SIZE, MONTHS, AND MANMONTHS FOR SYSTEM  1.
> SPERR1 HOL,46068,33,457
```

SYSTEM NAME	SIZE	DEV. TIME (MONTHS)	DEV. EFFORT (MANMONTHS)	LEVEL	TECHNOLOGY FACTOR
SPERR1 HOL	46068.	33.0	457.0	1	6

NEW W/ INT.

$C_K = 2584$

AVERAGE TECHNOLOGY FACTOR IS 6.

****

****

SUMMARY OF INPUT PARAMETERS

****

SYSTEM: SPERRY UNIVAC PROG 1 DATE: 16-Nov-79

PROJECT START: 1075

COST ELEMENTS

$/MY	50000.	INFLATION RATE	.065
STD DEV ($/MY)	5000.		

ENVIRONMENT

ONLINE DEV	0.40	HOL USAGE	0.38
DEVELOPMENT TIME	1.00	PRODUCTION TIME	0.00
LANGUAGE	JOVIAL		

SYSTEM

TYPE	COMMAND & CONTROL	REAL TIME CODE	0.50
LEVEL	1	UTILIZATION	0.50

MODERN PROGRAMMING PRACTICES

STRUCTURED PROG	1	DESIGN/CODE INSP	2
TOP-DOWN DEVELOPMENT	2	CHIEF PROGRAMMER TEAMS	1

EXPERIENCE

OVERALL	2	SYSTEM TYPE	1
LANGUAGE	1	HARDWARE	2

TECHNOLOGY FACTOR 6

SIZE

LOW	29913.	HIGH	62223.

HOL EQUIV.

****

****

SIMULATION

****

TITLE: SPERRY UNIVAC PROG 1 DATE: 14-Nov-79

HOL EQUIV.

*** SIMULATION RUNNING - PLEASE WAIT ***

	MEAN	STD DEV
SYSTEM SIZE (STMTS)	46068.	5385.
MINIMUM DEVELOPMENT TIME (MONTHS)	31.0	1.7
DEVELOPMENT EFFORT (MANMONTHS)	536.7	92.8
DEVELOPMENT COST (X $1000) (UNINFLATED DOLLARS)	2234.	437.
(INFLATED DOLLARS)	2423.	474.

SENSITIVITY PROFILE FOR MINIMUM TIME SOLUTION
(EXPECTED VALUES OF TIME, EFFORT, AND COST FOR VARIOUS SYSTEM SIZES)

	SOURCE STMTS	MONTHS	MANMONTHS	COST (X $1000)
(-3 SD)	29913.	25.8	246.	1025.
(-1 SD)	40683.	29.4	439.	1829.
MOST LIKELY	46068.	31.0	537.	2234.
(+1 SD)	51453.	32.6	643.	2680.
(+3 SD)	62223.	35.3	853.	3553.

A CONSISTENCY CHECK WITH DATA FROM OTHER SYSTEMS OF THE SAME SIZE SHOWS:

TOTAL MANMONTHS (537.)	WITHIN NORMAL RANGE
PROJECT DURATION (31.0 MONTHS)	WITHIN NORMAL RANGE
AVG # PEOPLE(17.)	WITHIN NORMAL RANGE
PRODUCTIVITY (86. LINES/MM)	WITHIN NORMAL RANGE

****

****

LINEAR PROGRAM

****

TITLE: SPERRY UNIVAC PROG 1 DATE: 14-Nov-79

THIS FUNCTION USES THE TECHNIQUE OF LINEAR PROGRAMMING (SIMPLEX ALGORITHM) TO DETERMINE THE MINIMUM EFFORT (AND COST) OR THE MINIMUM TIME IN WHICH A SYSTEM CAN BE BUILT. THE RESULTS ARE BASED ON THE ACTUAL MANPOWER, COST, AND SCHEDULE CONSTRAINTS OF THE USER, COMBINED WITH THE SYSTEM CONSTRAINTS YOU HAVE PROVIDED EARLIER TO YIELD A CONSTRAINED OPTIMAL SOLUTION.

ENTER THE MAXIMUM DEVELOPMENT COST IN DOLLARS> 2250000

ENTER MAXIMUM DEVELOPMENT TIME IN MONTHS> 36

ENTER THE MINIMUM AND MAXIMUM NUMBER OF PEOPLE YOU CAN HAVE ON BOARD AT PEAK MANLOADING TIME> 10,30

	TIME	EFFORT	COST (X $1000)
MINIMUM COST	36.0 MONTHS	299. MM	1246.
MINIMUM TIME	31.1 MONTHS	540. MM	2250.

YOUR REALISTIC TRADE-OFF REGION LIES BETWEEN THE LIMITS OF THE TABLE ABOVE.

(INTERPOLATION IN THE TRADE-OFF TABLE BETWEEN THESE LIMITS WILL PRODUCE ALL ACCEPTABLE ALTERNATIVES. WOULD YOU LIKE TO SEE A TRADE-OFF ANALYSIS WITHIN THESE LIMITS (Y OR N) ? Y

TIME	MANMONTHS	COST (X $1000)
31.1	540.	2250.
32.1	476.	1982.
33.1	421.	1753.
34.1	373.	1556.
35.1	333.	1386.
36.0	299.	1246.

THE RESULTS SHOWN IN THIS TABLE CAN BE USED WITH DESIGN-TO-COST OR NEW TIME TO GENERATE AN UPDATED FILE AND AN ENTIRELY NEW ARRAY OF CONSEQUENT RESULTS FOR MANLOADING, CASHFLOW, LIFE CYCLE, RISK ANALYSIS, COMPUTER TIME AND FRONT END ESTIMATES.

****

****

DESIGN TO COST

****

TITLE: SPERRY UNIVAC PROG 1 DATE: 14-Nov-79

SLIM HAS PROVIDED ITS BEST ESTIMATE OF THE MINIMUM TIME AND CORRESPONDING MAXIMUM EFFORT (AND COST) TO DEVELOP YOUR SYSTEM. THESE VALUES ARE:

MINIMUM TIME:		31.0 MONTHS
EFFORT:		537. MANMONTHS
COST (X $1000):	$	2236.

A GREATER EFFORT (OR COST) WOULD RESULT IN A VERY RISKY TIME SCHEDULE. HOWEVER, IF A LOWER EFFORT IS SPECIFIED (WITHIN REASONABLE LIMITS), DEVELOPMENT IS STILL FEASIBLE AS LONG AS YOU CAN TAKE MORE TIME.

ENTER DESIRED EFFORT IN MANMONTHS> 457

ACTUAL

		MEAN	STD DEV
NEW DEVELOPMENT TIME (MONTHS)		32.4	1.8
NEW DEVELOPMENT COST (X $1000)	$	1904.	329.

YOUR FILE IS UPDATED WITH THESE NEW PARAMETERS. RUN MANLOADING AND CASHFLOW OR LIFE CYCLE TO SEE HOW THESE SAVINGS CAN BE REALIZED.

A CONSISTENCY CHECK WITH DATA FROM OTHER SYSTEMS OF THE SAME SIZE SHOWS:

TOTAL MANMONTHS (457.)	WITHIN NORMAL RANGE
PROJECT DURATION (32.4 MONTHS)	LONGER THAN NORMAL TIME DURATION
AVG # PEOPLE(14.)	WITHIN NORMAL RANGE
PRODUCTIVITY (101. LINES/MM)	WITHIN NORMAL RANGE

****

LIFE CYCLE

SYSTEM: SPERRY UNIVAC PROG 1 DATE: 14-Nov-79

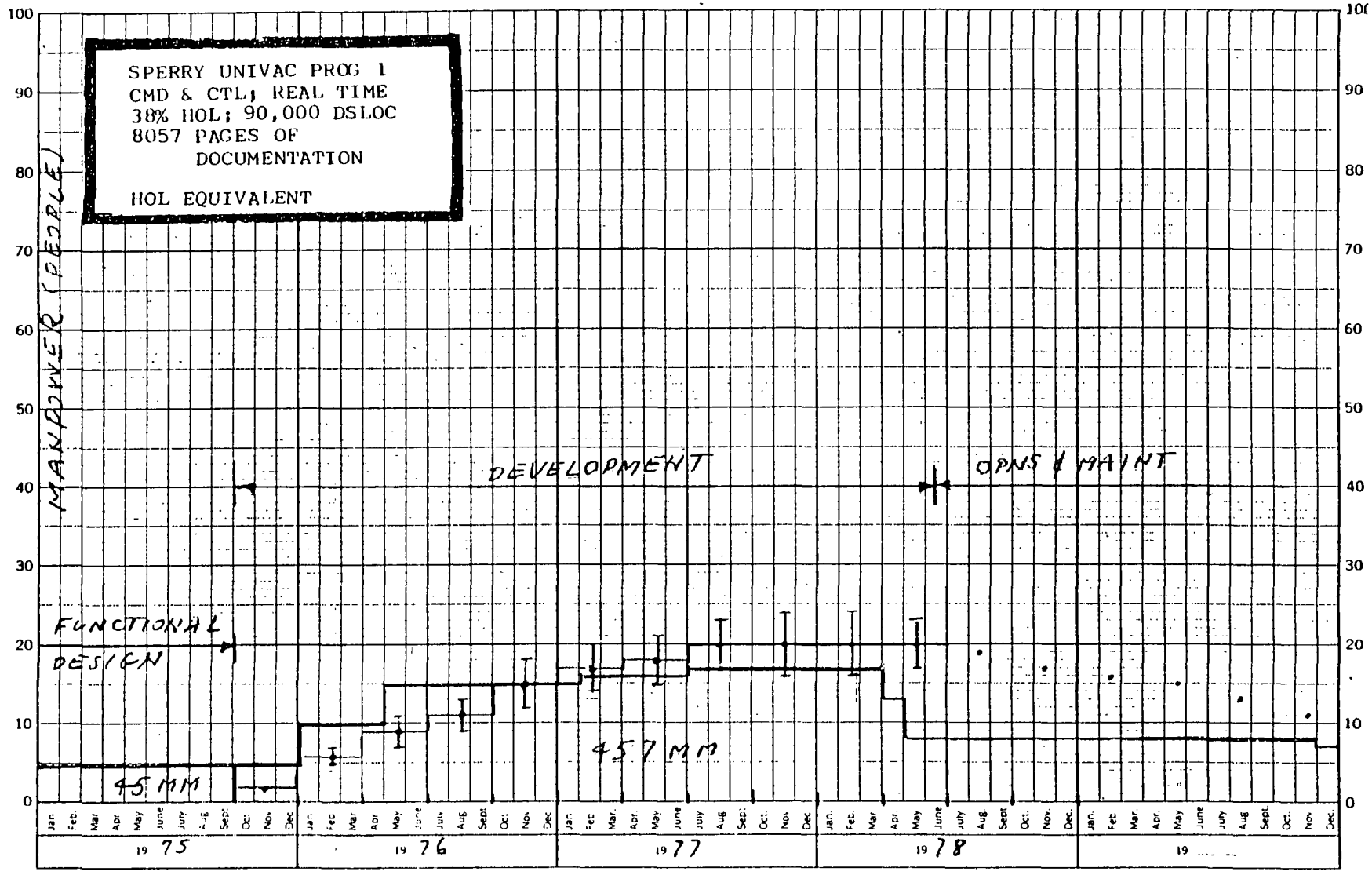
HOL EQUIV.

THE TABLE BELOW SHOWS THE MEAN PROJECTED EFFORT AND CASHFLOW (AND ASSOCIATED STANDARD DEVIATIONS) OVER THE LIFE CYCLE OF THE SYSTEM. ALL PROJECTIONS ARE BASED ON AN OPTIMAL APPLICATION OF RESOURCES OVER TIME. THE INPUT PARAMETERS ARE:

	MEAN	STD DEV
DEVELOPMENT TIME (MONTHS)	32.4	1.8
LIFE CYCLE EFFORT(MM)	888.6	153.7
AVG COST/MY (X $1000)	50.	5.
INFLATION RATE	0.065	0.010

QTR ENDING	PEOPLE		COST/QTR (X $1000)		CUM COST (X $1000)	
	MEAN	STD DEV	MEAN	STD DEV	MEAN	STD DEV
DEC 75	2.	0.	23.	6.	23.	5.
MAR 76	6.	1.	71.	16.	94.	18.
JUN 76	9.	2.	117.	26.	211.	41.
SEP 76	11.	2.	153.	36.	366.	72.
DEC 76	15.	3.	196.	47.	559.	109.
MAR 77	17.	3.	231.	52.	787.	154.
JUN 77	18.	3.	248.	55.	1039.	203.
SEP 77	20.	3.	271.	51.	1307.	256.
DEC 77	20.	4.	281.	58.	1588.	311.
MAR 78	20.	4.	286.	59.	1873.	366.
JUN 78	20.	3.	290.	57.	2159.	422.
SEP 78	19.	3.	284.	54.	2441.	477.
DEC 78	17.	3.	265.	55.	2708.	530.
MAR 79	16.	3.	244.	56.	2953.	578.
JUN 79	15.	3.	233.	50.	3181.	622.
SEP 79	13.	3.	204.	45.	3388.	663.
DEC 79	11.	2.	183.	36.	3569.	698.
MAR 80	9.	2.	158.	31.	3728.	729.
JUN 80	8.	2.	134.	32.	3863.	756.
SEP 80	7.	2.	113.	31.	3977.	778.
DEC 80	5.	1.	95.	25.	4072.	796.
MAR 81	5.	1.	82.	25.	4152.	812.
JUN 81	4.	1.	64.	20.	4218.	825.
SEP 81	3.	1.	51.	17.	4269.	835.
DEC 81	2.	1.	39.	15.	4309.	843.
MAR 82	2.	1.	32.	12.	4340.	849.
JUN 82	1.	1.	26.	11.	4366.	854.
SEP 82	1.	0.	19.	10.	4386.	858.
DEC 82	1.	0.	15.	7.	4401.	861.
MAR 83	1.	0.	10.	5.	4411.	863.

LIFE CYCLE PROJECTIONS


SPERRY UNIVAC PROG 1
CMD & CTL; REAL TIME
38% HOL; 90,000 DSLOC
8057 PAGES OF
DOCUMENTATION
HOL EQUIVALENT
MANPOWER (PEOPLE)
DEVELOPMENT
OPNS & MAINT
FUNCTIONAL
DESIGN
45 MM
457 MM
100
90
80
70
60
50
40
30
20
10
0
Jan Feb Mar Apr May June July Aug Sept Oct Nov Dec
19 75
19 76
19 77
19 78

****

RISK ANALYSIS

****

TITLE: SPERRY UNIVAC PROG 1 DATE: 14-Nov-79

THE TABLES BELOW SHOW THE PROBABILITY THAT IT WILL NOT TAKE MORE THAN THE INDICATED AMOUNT OF TIME, EFFORT, AND DOLLARS TO DEVELOP YOUR SYSTEM.

PROBABILITY	TIME (MONTHS)
1. %	28.2
5. %	29.4
10. %	30.1
20. %	30.8
30. %	31.4
40. %	31.9
50. %	32.4
60. %	32.8
70. %	33.3
80. %	33.9
90. %	34.7
95. %	35.4
99. %	36.6

EXPECTED

PROBABILITY PROFILE

PROBABILITY	MANMONTHS	COST (X $1000)	INFLATED COST(X $1000)
1. %	273.	1138.	1239.
5. %	327.	1362.	1483.
10. %	356.	1482.	1613.
20. %	390.	1627.	1771.
30. %	416.	1732.	1885.
40. %	437.	1821.	1982.
50. %	457.	1904.	2073.
60. %	477.	1987.	2164.
70. %	498.	2077.	2261.
80. %	524.	2181.	2375.
90. %	558.	2326.	2533.
95. %	587.	2446.	2663.
99. %	641.	2671.	2907.

PROBABILITY PROFILE

****

DOCUMENTATION

****

TITLE: SPERRY UNIVAC PROG 1 DATE: 14-Nov-79

IT IS POSSIBLE TO ESTIMATE THE NUMBER OF PAGES OF DOCUMENTATION, BASED ON DATA COLLECTED FROM SEVERAL HUNDRED SYSTEMS.

THE EXPECTED NUMBER FOR YOUR SYSTEM IS 3224 PAGES.

THE 90% RANGE IS FROM 921 TO 7831 PAGES.

****

ACTUAL: 8057

****

CALIBRATE

****

.HIS FUNCTION ENABLES THE USER TO MAKE FUTURE ESTIMATES BASED ON HISTORICAL DATA FROM HIS ORGANIZATION AS WELL AS ON THE TYPE AND SIZE OF THE SYSTEM. IN ESSENCE, **CALIBRATE** TAKES TIME AND MANPOWER DATA FROM PAST SOFTWARE PROJECTS AND COMPUTES A TECHNOLOGY FACTOR FOR THE USER' ^O...
1

ENTER ALL DATA FOR EACH SYSTEM ON 1 LINE, SEPARATED BY COMMAS.

ENTER SYSTEM NAME, SIZE, MONTHS, AND MANMONTHS FOR SYSTEM 1.
> SPERRY UNIVAC 2,500000,51,2682

SYSTEM NAME	SIZE	DEV. TIME (MONTHS)	DEV. EFFORT (MANMONTHS)	LEVEL	TECHNOLOGY FACTOR
SPERRY UNIVAC 2	500000.	51.0	2682.0	1	11

NEW W/ INT.

$C_K = 8362$

AVERAGE TECHNOLOGY FACTOR IS 11.

```
**********************************************************************************
                          SUMMARY OF INPUT PARAMETERS
**********************************************************************************
SYSTEM:  SPERRY UNIVAC PROGV 2                                  DATE: 13-Aug-79


PROJECT START:  175

COST ELEMENTS
  $/MY                50000.                      INFLATION RATE  .065
  STD DEV ($/MY)       5000.

ENVIRONMENT
  ONLINE DEV           0.90                            HOL USAGE  0.99
  DEVELOPMENT TIME     1.00                    PRODUCTION TIME  0.10
  LANGUAGE           JOVIAL

SYSTEM
  TYPE  COMMAND & CONTROL                        REAL TIME CODE  0.05
  LEVEL                  1                          UTILIZATION  0.50

MODERN PROGRAMMING PRACTICES
  STRUCTURED PROG        1                    DESIGN/CODE INSP      1
  TOP-DOWN DEVELOPMENT   1                CHIEF PROGRAMMER TEAMS    1

EXPERIENCE
  OVERALL                2                       SYSTEM TYPE      2
  LANGUAGE               2                          HARDWARE      2

TECHNOLOGY
  FACTOR                11

SIZE
  LOW               450000.                            HIGH  550000.


**********************************************************************************
```

****

SIMULATION

****

TITLE: SPERRY UNIVAC PROG 2 DATE: 13-Aug-79

*** SIMULATION RUNNING - PLEASE WAIT ***

	MEAN	STD DEV
SYSTEM SIZE (STMTS)	500000.	16667.
MINIMUM DEVELOPMENT TIME (MONTHS)	52.3	1.4
DEVELOPMENT EFFORT (MANMONTHS)	2811.1	274.9
DEVELOPMENT COST (X $1000) (UNINFLATED DOLLARS)	11688.	1654.
(INFLATED DOLLARS)	13406.	1914.

SENSITIVITY PROFILE FOR MINIMUM TIME SOLUTION
(EXPECTED VALUES OF TIME, EFFORT, AND COST FOR VARIOUS SYSTEM SIZES)

	SOURCE STMTS	MONTHS	MANMONTHS	COST (X $1000)
(-3 SD)	450000.	49.9	2471.	10295.
(-1 SD)	483333.	51.4	2709.	11286.
MOST LIKELY	500000.	52.3	2811.	11688.
(+1 SD)	516667.	52.9	2951.	12296.
(+3 SD)	550000.	54.3	3198.	13326.

CONSISTENCY CHECK WITH DATA FROM OTHER SYSTEMS OF THE SAME SIZE SHOWS:

TOTAL MANMONTHS (2811.)	WITHIN NORMAL RANGE
PROJECT DURATION (52.3 MONTHS)	WITHIN NORMAL RANGE
AVG # PEOPLE(54.)	WITHIN NORMAL RANGE
PRODUCTIVITY (178. LINES/MM)	WITHIN NORMAL RANGE

****

****

RISK ANALYSIS

****

TITLE: SPERRY UNIVAC PROG 2 DATE: 13-Aug-79

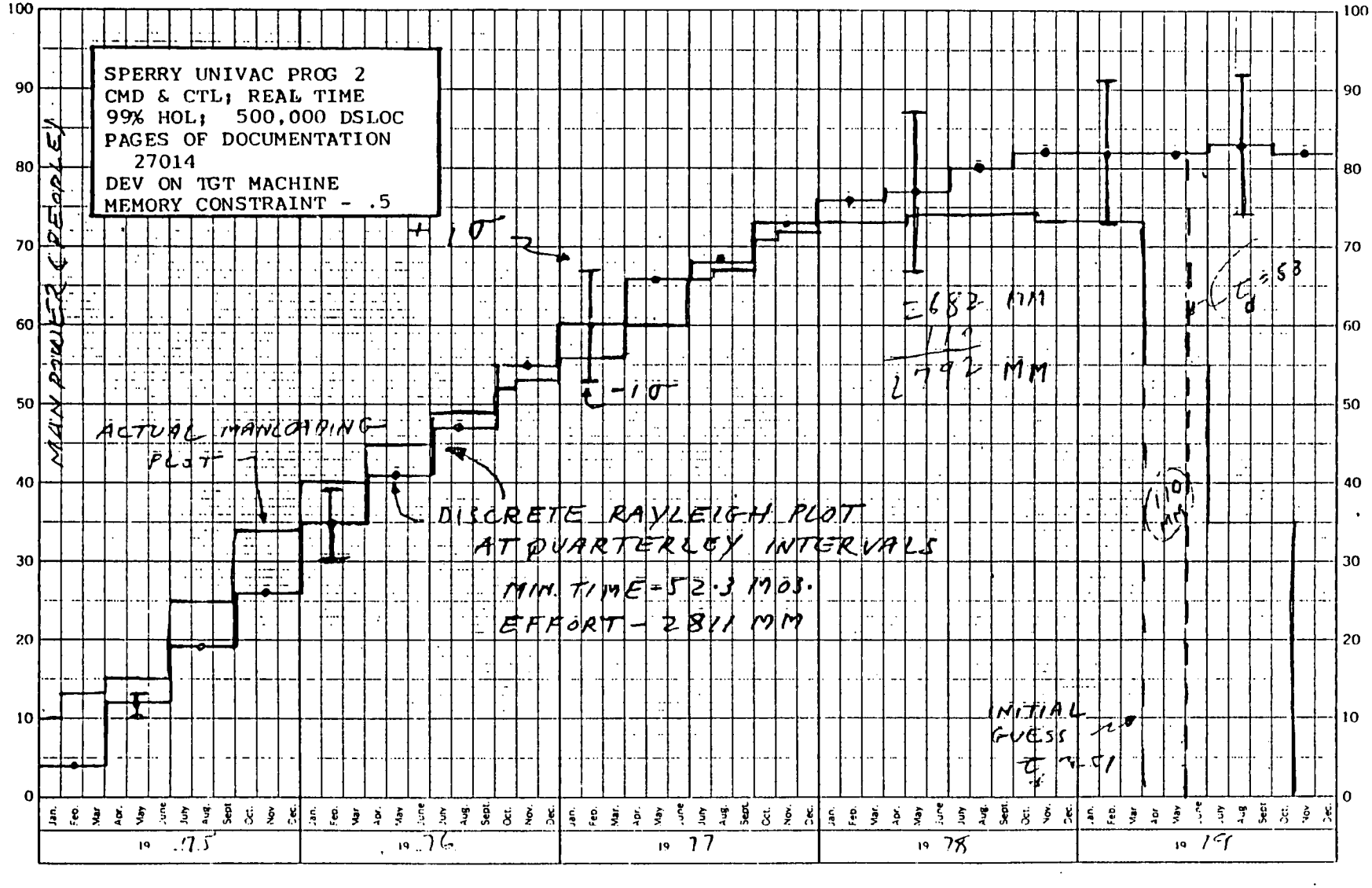
THE TABLES BELOW SHOW THE PROBABILITY THAT IT WILL NOT TAKE MORE THAN THE INDICATED AMOUNT OF TIME, EFFORT, AND DOLLARS TO DEVELOP YOUR SYSTEM.

PROBABILITY	TIME (MONTHS)
1. %	49.0
5. %	50.0
10. %	50.5
20. %	51.1
30. %	51.5
40. %	51.9
EXPECTED → 50. %	52.3
60. %	52.6
70. %	53.0
80. %	53.5
90. %	54.1
95. %	54.6
99. %	55.5

PROBABILITY PROFILE

PROBABILITY	MANMONTHS	COST (X $1000)	INFLATED COST(X $1000)
1. %	2171.	9047.	10378.
5. %	2359.	9828.	11273.
10. %	2459.	10245.	11751.
20. %	2580.	10749.	12329.
30. %	2667.	11113.	12746.
40. %	2742.	11423.	13103.
→ 50. %	2811.	11713.	13435.
60. %	2881.	12003.	13767.
70. %	2955.	12313.	14123.
80. %	3042.	12677.	14541.
90. %	3163.	13181.	15119.
95. %	3263.	13597.	15597.
99. %	3451.	14378.	16492.

PROBABILITY PROFILE


SPERRY UNIVAC PROG 2
CMD & CTL; REAL TIME
99% HOL; 500,000 DSLOC
PAGES OF DOCUMENTATION
27014
DEV ON TGT MACHINE
MEMORY CONSTRAINT - .5
MAN POWER (PEOPLE)
+1σ
-1σ
ACTUAL MANLOADING PLOT
DISCRETE RAYLEIGH PLOT AT QUARTERLY INTERVALS
MIN. TIME=52.3 MOS.
EFFORT - 2811 MM
E682 MM
2792 MM
INITIAL GUESS
100
90
80
70
60
50
40
30
20
10
0
Jan. Feb. Mar. Apr. May June July Aug. Sept. Oct. Nov. Dec.
19 75
19 76
19 77
19 78
19 79

```
AVAILABLE FUNCTIONS ARE:
CALIBRATE
EDITOR
ESTIMATE
BYE .

FUNCTION? EST

INPUT FILENAME? SPERR3


INPUT DATA CHECK - OK
```

```
**********************************************************************************
                                    CALIBRATE
**********************************************************************************

THIS FUNCTION ENABLES THE USER TO MAKE FUTURE ESTIMATES BASED ON HISTORICAL
DATA FROM HIS ORGANIZATION AS WELL AS ON THE TYPE AND SIZE OF THE SYSTEM.  IN
ESSENCE, **CALIBRATE** TAKES TIME AND MANPOWER DATA FROM PAST SOFTWARE
PROJECTS AND COMPUTES A TECHNOLOGY FACTOR FOR THE USER'S  ORGANIZATION.  THIS
FACTOR IS REALLY AN INDICATION OF THE STATE OF TECHNOLOGY WHICH A PARTICULAR
ORGANIZATION APPLIES TO A SOFTWARE PROJECT.

THE FOLLOWING HISTORICAL DATA IS REQUIRED:
  (1) SYSTEM NAME (UP TO 20 CHARACTERS)
  (2) TOTAL SYSTEM SIZE IN SOURCE STATEMENTS
  (3) NUMBER OF MONTHS TO DEVELOP
  (4) NUMBER OF MANMONTHS TO DEVELOP

HISTORICAL DATA WILL BE PROVIDED FOR HOW MANY SYSTEMS? 1

ENTER ALL DATA FOR EACH SYSTEM ON 1 LINE, SEPARATED BY COMMAS.


ENTER SYSTEM NAME, SIZE, MONTHS, AND MANMONTHS FOR SYSTEM  1.
> SPERR3,16724,26,399


.....................................................................................
SYSTEM NAME              SIZE    DEV. TIME    DEV. EFFORT    LEVEL   TECHNOLOGY
                                 (MONTHS)     (MANMONTHS)             FACTOR
.....................................................................................

SPERR3                 16724.      26.0         399.0         3          2

.....................................................................................


                AVERAGE TECHNOLOGY FACTOR IS     2.


**********************************************************************************
```

SUMMARY OF INPUT DATA PRINTED (Y OR N)? Y

```
**********************************************************************
                        SUMMARY OF INPUT PARAMETERS
**********************************************************************
SYSTEM:  SPERRY UNIVAC 3                                  DATE: 16-Nov-79


PROJECT START:  175

COST ELEMENTS
  $/MY                50000.                    INFLATION RATE  .070
  STD DEV ($/MY)       5000.

ENVIRONMENT
  ONLINE DEV           0.25                          HOL USAGE  0.53
  DEVELOPMENT TIME     0.20                 PRODUCTION TIME     0.80
  LANGUAGE           JOVIAL

SYSTEM
  TYPE  COMMAND & CONTROL                     REAL TIME CODE     0.20
  LEVEL                 2                        UTILIZATION     0.52

MODERN PROGRAMMING PRACTICES
  STRUCTURED PROG       2                 DESIGN/CODE INSP          2
  TOP-DOWN DEVELOPMENT  3           CHIEF PROGRAMMER TEAMS        2

EXPERIENCE
  OVERALL               3                       SYSTEM TYPE       2
  LANGUAGE              3                          HARDWARE       3

TECHNOLOGY
  FACTOR                5

SIZE
  LOW               20600.                             HIGH   32600.


**********************************************************************
```

****

SIMULATION

****

TITLE: SPERRY UNIVAC 3 DATE: 16-Nov-79

*** SIMULATION RUNNING - PLEASE WAIT ***

	MEAN	STD DEV
SYSTEM SIZE (STMTS)	16724.	776.
MINIMUM DEVELOPMENT TIME (MONTHS)	25.3	0.8
DEVELOPMENT EFFORT (MANMONTHS)	471.6	51.7
DEVELOPMENT COST (X $1000) (UNINFLATED DOLLARS)	1961.	290.
(INFLATED DOLLARS)	2106.	312.

SENSITIVITY PROFILE FOR MINIMUM TIME SOLUTION
(EXPECTED VALUES OF TIME, EFFORT, AND COST FOR VARIOUS SYSTEM SIZES)

	SOURCE STMTS	MONTHS	MANMONTHS	COST (X $1000)
(-3 SD)	14396.	23.6	391.	1630.
(-1 SD)	15948.	24.7	446.	1859.
MOST LIKELY	16724.	25.3	472.	1961.
(+1 SD)	17500.	25.7	503.	2095.
(+3 SD)	19052.	26.7	605.	2520.

A CONSISTENCY CHECK WITH DATA FROM OTHER SYSTEMS OF THE SAME SIZE SHOWS:

TOTAL MANMONTHS (472.)	GREATER THAN NORMAL EFFORT
PROJECT DURATION (25.3 MONTHS)	LONGER THAN NORMAL TIME DURATION
AVG # PEOPLE (19.)	GREATER THAN NORMAL # OF PEOPLE
PRODUCTIVITY (35. LINES/MM)	LESS THAN NORMAL PRODUCTIVITY

****

```
AVAILABLE OPTIONS ARE:

NEW TIME              LINEAR PROGRAM      RISK ANALYSIS          DOCUMENTATION
DESIGN-TO-COST        MANLOADING          BENEFIT ANALYSIS       ALL ANALYSES
PERT SIZING           CASHFLOW            MILESTONES             HELP
DESIGN-TO-RISK(DTR)   LIFE CYCLE          CPU USAGE              END
FRONT-END ESTIMATES

OPTION? LIN


**********************************************************************************
                                LINEAR PROGRAM
**********************************************************************************
TITLE:  SPERRY UNIVAC 3                                          DATE: 16-Nov-79

   THIS FUNCTION USES THE TECHNIQUE OF LINEAR PROGRAMMING (SIMPLEX ALGORITHM)
   TO DETERMINE THE MINIMUM EFFORT (AND COST) OR THE MINIMUM TIME IN WHICH
   A SYSTEM CAN BE BUILT.  THE RESULTS ARE BASED ON THE ACTUAL MANPOWER, COST,
   AND SCHEDULE CONSTRAINTS OF THE USER, COMBINED WITH THE SYSTEM CONSTRAINTS
   YOU HAVE PROVIDED EARLIER TO YIELD A CONSTRAINED OPTIMAL SOLUTION.

ENTER THE MAXIMUM DEVELOPMENT COST IN DOLLARS> 2000000

ENTER MAXIMUM DEVELOPMENT TIME IN MONTHS> 30

ENTER THE MINIMUM AND MAXIMUM NUMBER OF PEOPLE YOU
CAN HAVE ON BOARD AT PEAK MANLOADING TIME> 15,30
```

	TIME	EFFORT	COST (X $1000)
MINIMUM COST	28.8 MONTHS	277. MM	1153.
MINIMUM TIME	25.3 MONTHS	471. MM	1962.

YOUR REALISTIC TRADE-OFF REGION LIES BETWEEN THE LIMITS OF THE TABLE ABOVE.

(INTERPOLATION IN THE TRADE-OFF TABLE BETWEEN THESE LIMITS WILL PRODUCE ALL ACCEPTABLE ALTERNATIVES. WOULD YOU LIKE TO SEE A TRADE-OFF ANALYSIS WITHIN THESE LIMITS (Y OR N) ? Y

TIME	MANMONTHS	COST (X $1000)
25.3	471.	1962.
26.3	403.	1680.
27.3	347.	1447.
28.3	301.	1253.
28.8	277.	1153.

THE RESULTS SHOWN IN THIS TABLE CAN BE USED WITH DESIGN-TO-COST OR NEW TIME TO GENERATE AN UPDATED FILE AND AN ENTIRELY NEW ARRAY OF CONSEQUENT RESULTS FOR MANLOADING, CASHFLOW, LIFE CYCLE, RISK ANALYSIS, COMPUTER TIME AND FRONT END ESTIMATES.

****

****

DESIGN TO COST

****

TITLE: SPERRY UNIVAC 3 DATE: 16-Nov-79

SLIM HAS PROVIDED ITS BEST ESTIMATE OF THE MINIMUM TIME AND CORRESPONDING MAXIMUM EFFORT (AND COST) TO DEVELOP YOUR SYSTEM. THESE VALUES ARE:

MINIMUM TIME:		25.3 MONTHS
EFFORT:		472. MANMONTHS
COST (X $1000):	$	1965.

A GREATER EFFORT (OR COST) WOULD RESULT IN A VERY RISKY TIME SCHEDULE. HOWEVER, IF A LOWER EFFORT IS SPECIFIED (WITHIN REASONABLE LIMITS), DEVELOPMENT IS STILL FEASIBLE AS LONG AS YOU CAN TAKE MORE TIME.

ENTER DESIRED EFFORT IN MANMONTHS> 399

		MEAN	STD DEV
NEW DEVELOPMENT TIME (MONTHS)		26.3	0.8
NEW DEVELOPMENT COST (X $1000)	$	1663.	182.

YOUR FILE IS UPDATED WITH THESE NEW PARAMETERS. RUN MANLOADING AND CASHFLOW OR LIFE CYCLE TO SEE HOW THESE SAVINGS CAN BE REALIZED.

A CONSISTENCY CHECK WITH DATA FROM OTHER SYSTEMS OF THE SAME SIZE SHOWS:

TOTAL MANMONTHS (399.)	GREATER THAN NORMAL EFFORT
PROJECT DURATION (26.3 MONTHS)	LONGER THAN NORMAL TIME DURATION
AVG # PEOPLE (15.)	GREATER THAN NORMAL # OF PEOPLE
PRODUCTIVITY (42. LINES/MM)	LESS THAN NORMAL PRODUCTIVITY

****

****

RISK ANALYSIS

****

TITLE: SPERRY UNIVAC 3 DATE: 16-Nov-79

THE TABLES BELOW SHOW THE PROBABILITY THAT IT WILL NOT TAKE MORE THAN THE INDICATED AMOUNT OF TIME, EFFORT, AND DOLLARS TO DEVELOP YOUR SYSTEM.

PROBABILITY	TIME (MONTHS)
1. %	24.5
5. %	25.0
10. %	25.3
20. %	25.7
30. %	25.9
40. %	26.1
50. %	26.3
60. %	26.5
70. %	26.7
80. %	27.0
90. %	27.3
95. %	27.6
99. %	28.2

PROBABILITY PROFILE

PROBABILITY	MANMONTHS	COST (X $1000)	INFLATED COST(X $1000)
1. %	297.	1239.	1334.
5. %	327.	1363.	1468.
10. %	343.	1429.	1539.
20. %	362.	1509.	1625.
30. %	376.	1567.	1688.
40. %	388.	1616.	1741.
50. %	399.	1663.	1791.
60. %	410.	1709.	1840.
70. %	422.	1758.	1893.
80. %	436.	1816.	1956.
90. %	455.	1896.	2042.
95. %	471.	1962.	2113.
99. %	501.	2086.	2247.

PROBABILITY PROFILE

****

MANLOADING

****

TITLE: SPERRY UNIVAC 3 DATE: 16-Nov-79

THE TABLE BELOW SHOWS THE MEAN PROJECTED EFFORT
AND ASSOCIATED (+ OR -) STANDARD DEVIATION REQUIRED
FOR DEVELOPMENT. THE INPUT PARAMETERS ARE:

	MEAN	STD DEV
DEVELOPMENT EFFORT (MM)	399.0	41.4
DEVELOPMENT TIME (MONTHS)	26.3	0.8

*** SIMULATION RUNNING - PLEASE WAIT ***

TIME	PEOPLE/MONTH	STD DEV	CUMULATIVE MANMONTHS	CUM STD DEV
JAN 75	1.83	0.22	2.	0.
FEB 75	5.38	0.63	7.	1.
MAR 75	8.88	1.06	16.	2.
APR 75	12.17	1.44	28.	3.
MAY 75	14.99	1.78	43.	4.
JUN 75	17.61	2.07	61.	6.
JUL 75	19.71	2.17	80.	8.
AUG 75	21.53	2.50	102.	11.
SEP 75	22.63	2.52	124.	13.
OCT 75	23.35	2.55	148.	15.
NOV 75	23.54	2.60	171.	18.
DEC 75	23.61	2.55	195.	20.
JAN 76	23.05	2.48	218.	23.
FEB 76	22.24	2.40	240.	25.
MAR 76	21.15	2.12	261.	27.
APR 76	19.93	2.07	281.	29.
MAY 76	18.43	1.99	299.	31.
JUN 76	16.89	1.76	316.	33.
JUL 76	15.18	1.71	331.	34.
AUG 76	13.66	1.50	345.	36.
SEP 76	11.94	1.38	357.	37.
OCT 76	10.44	1.22	368.	38.
NOV 76	9.12	1.13	377.	39.
DEC 76	7.78	1.02	384.	40.
JAN 77	6.63	0.95	391.	41.
FEB 77	5.56	0.86	397.	41.
MAR 77	2.30	0.37	399.	42.

****

****

SIMULATION

****

TITLE: SPERRY UNIVAC 3 DATE: 16-Nov-7

*** SIMULATION RUNNING - PLEASE WAIT ***

HOL EQUIV.

	MEAN	STD DEV
SYSTEM SIZE (STMTS)	16724.	776.
MINIMUM DEVELOPMENT TIME (MONTHS)	25.3	0.8
DEVELOPMENT EFFORT (MANMONTHS)	472.1	49.0
DEVELOPMENT COST (X $1000) (UNINFLATED DOLLARS)	1969.	288.
(INFLATED DOLLARS)	2114.	310.

SENSITIVITY PROFILE FOR MINIMUM TIME SOLUTION
(EXPECTED VALUES OF TIME, EFFORT, AND COST FOR VARIOUS SYSTEM SIZES)

	SOURCE STMTS	MONTHS	MANMONTHS	COST (X $1000)
(-3 SD)	14396.	23.6	391.	1630.
(-1 SD)	15948.	24.7	446.	1859.
MOST LIKELY	16724.	25.3	472.	1969.
(+1 SD)	17500.	25.7	503.	2095.
(+3 SD)	19052.	26.7	605.	2520.

A CONSISTENCY CHECK WITH DATA FROM OTHER SYSTEMS OF THE SAME SIZE SHOWS:

TOTAL MANMONTHS (472.)	GREATER THAN NORMAL EFFORT
PROJECT DURATION (25.3 MONTHS)	LONGER THAN NORMAL TIME DURATION
AVG # PEOPLE (19.)	GREATER THAN NORMAL # OF PEOPLE
PRODUCTIVITY (35. LINES/MM)	LESS THAN NORMAL PRODUCTIVITY

****

****

DESIGN TO COST

****

TITLE: SPERRY UNIVAC 3 DATE: 16-Nov-7

SLIM HAS PROVIDED ITS BEST ESTIMATE OF THE MINIMUM TIME AND CORRESPONDING MAXIMUM EFFORT (AND COST) TO DEVELOP YOUR SYSTEM. THESE VALUES ARE:

MINIMUM TIME:		25.3 MONTHS
EFFORT:		472. MANMONTHS
COST (X $1000):	$	1967.

A GREATER EFFORT (OR COST) WOULD RESULT IN A VERY RISKY TIME SCHEDULE. HOWEVER, IF A LOWER EFFORT IS SPECIFIED (WITHIN REASONABLE LIMITS), DEVELOPMENT IS STILL FEASIBLE AS LONG AS YOU CAN TAKE MORE TIME.

ENTER DESIRED EFFORT IN MANMONTHS> 399

		MEAN	STD DEV
NEW DEVELOPMENT TIME (MONTHS)		26.3	0.8
NEW DEVELOPMENT COST (X $1000)	$	1663.	173.

YOUR FILE IS UPDATED WITH THESE NEW PARAMETERS. RUN MANLOADING AND CASHFLOW OR LIFE CYCLE TO SEE HOW THESE SAVINGS CAN BE REALIZED.

A CONSISTENCY CHECK WITH DATA FROM OTHER SYSTEMS OF THE SAME SIZE SHOWS:

TOTAL MANMONTHS (399.)	GREATER THAN NORMAL EFFORT
PROJECT DURATION (26.3 MONTHS)	LONGER THAN NORMAL TIME DURATION
AVG # PEOPLE (15.)	GREATER THAN NORMAL # OF PEOPLE
PRODUCTIVITY (42. LINES/MM)	LESS THAN NORMAL PRODUCTIVITY

****

****

LIFE CYCLE

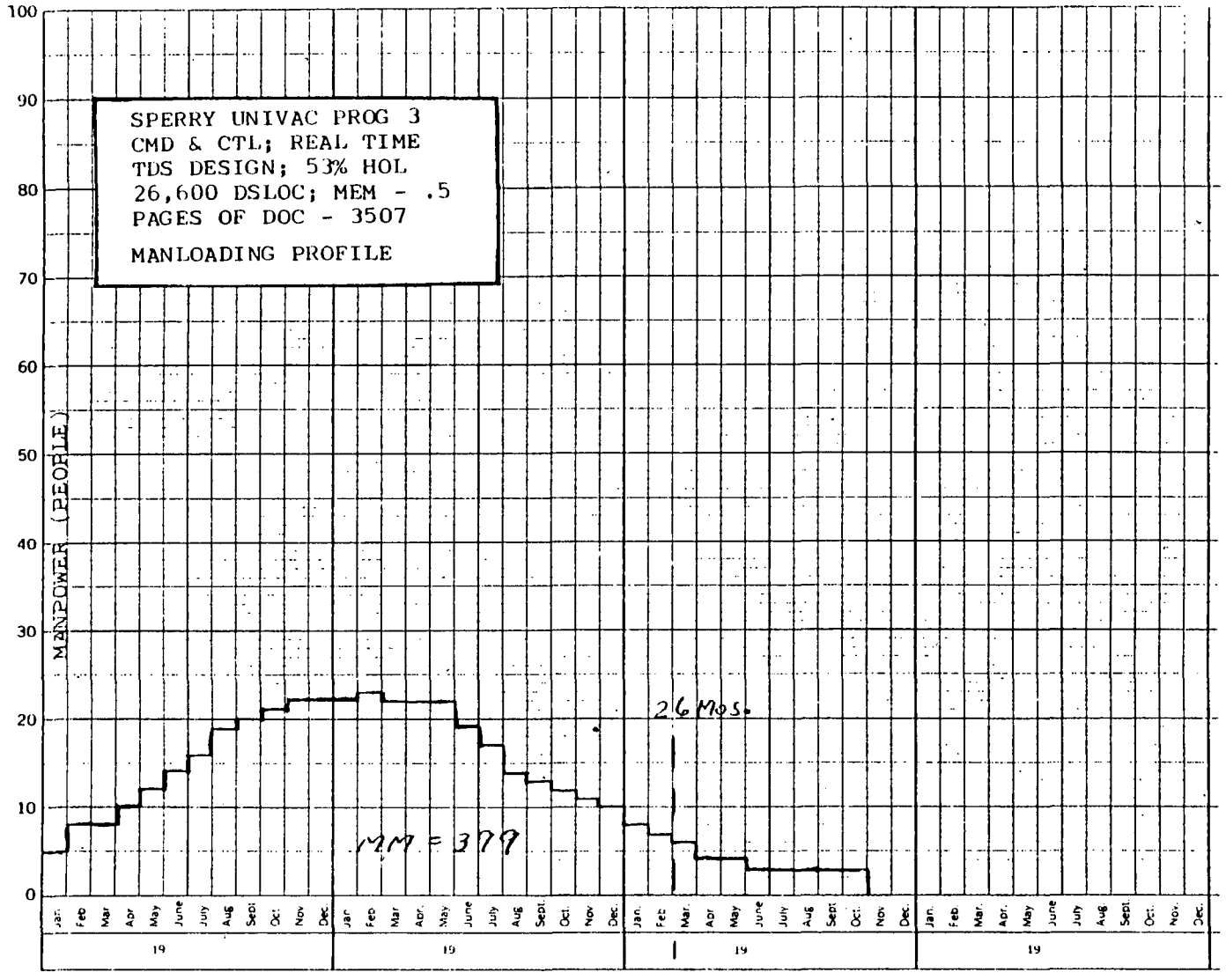
****

SYSTEM: SPERRY UNIVAC 3 DATE: 16-Nov-7

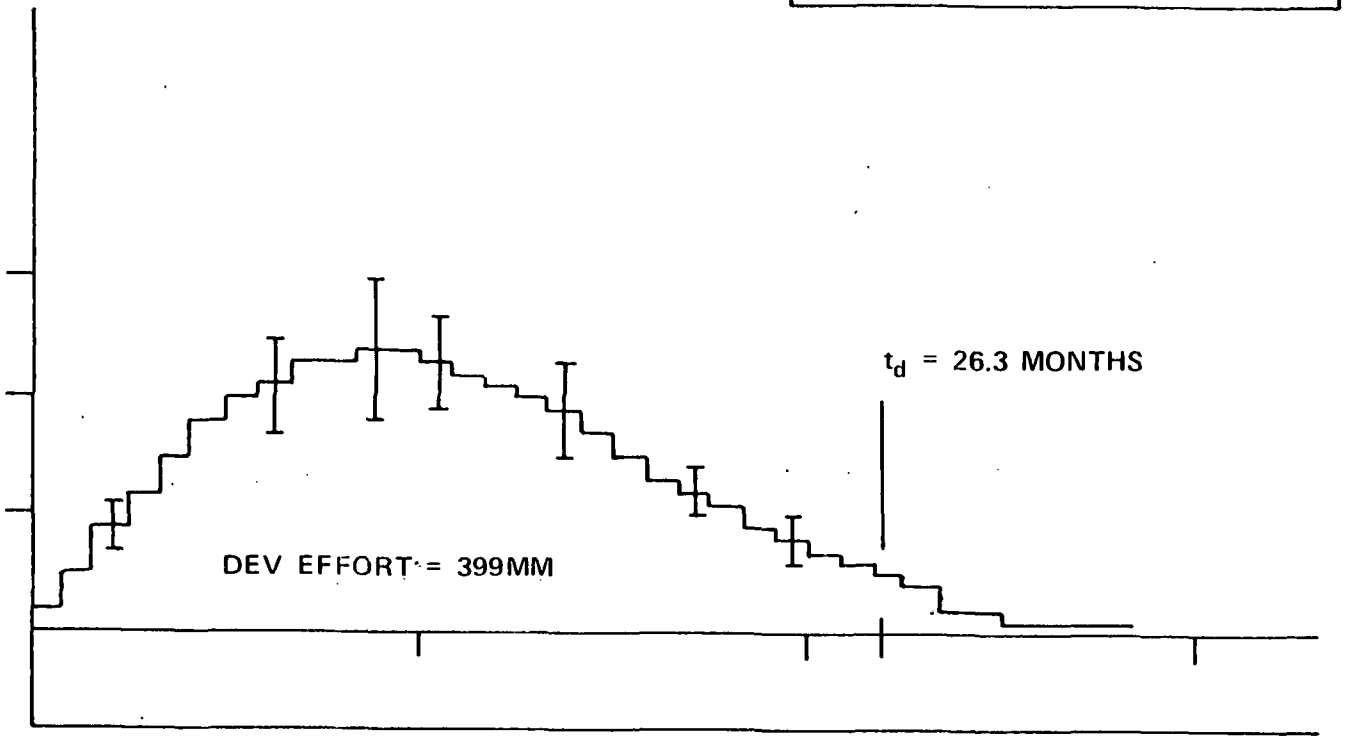
THE TABLE BELOW SHOWS THE MEAN PROJECTED EFFORT AND CASHFLOW (AND ASSOCIATED STANDARD DEVIATIONS) OVER THE LIFE CYCLE OF THE SYSTEM. ALL PROJECTIONS ARE BASED ON AN OPTIMAL APPLICATION OF RESOURCES OVER TIME. THE INPUT PARAMETERS ARE:

	MEAN	STD DEV
DEVELOPMENT TIME (MONTHS)	26.3	0.8
LIFE CYCLE EFFORT(MM)	419.9	43.6
AVG COST/MY (X $1000)	50.	5.
INFLATION RATE	0.070	0.011

MONTH	PEOPLE		COST/MTH (X $1000)		CUM COST (X $1000)	
	MEAN	STD DEV	MEAN	STD DEV	MEAN	STD DEV
JAN 75	2.	0.	8.	1.	8.	1.
FEB 75	5.	1.	23.	4.	30.	4.
MAR 75	9.	1.	38.	6.	68.	10.
APR 75	12.	1.	51.	8.	119.	17.
MAY 75	15.	2.	65.	11.	183.	27.
JUN 75	18.	2.	76.	12.	259.	38.
JUL 75	20.	2.	85.	14.	344.	50.
AUG 75	21.	2.	93.	14.	437.	64.
SEP 75	23.	3.	99.	15.	536.	79.
OCT 75	23.	3.	103.	16.	638.	94.
NOV 75	24.	3.	104.	16.	742.	109.
DEC 75	24.	3.	106.	15.	847.	124.
JAN 76	23.	2.	103.	15.	951.	140.
FEB 76	22.	2.	100.	15.	1051.	154.
MAR 76	21.	2.	95.	14.	1146.	168.
APR 76	20.	2.	91.	13.	1236.	181.
MAY 76	19.	2.	85.	12.	1321.	194.
JUN 76	17.	2.	77.	11.	1399.	205.
JUL 76	15.	2.	71.	10.	1469.	216.
AUG 76	14.	2.	63.	9.	1532.	225.
SEP 76	12.	1.	56.	9.	1589.	233.
OCT 76	11.	1.	50.	8.	1639.	240.
NOV 76	9.	1.	44.	7.	1682.	247.
DEC 76	8.	1.	37.	6.	1720.	252.
JAN 77	7.	1.	32.	5.	1752.	257.
FEB 77	6.	1.	27.	5.	1778.	261.
MAR 77	5.	1.	23.	4.	1801.	264.
APR 77	4.	1.	18.	4.	1820.	267.
MAY 77	3.	1.	15.	3.	1835.	269.
JUN 77	3.	0.	12.	3.	1847.	271.
JUL 77	2.	0.	10.	2.	1857.	272.
AUG 77	2.	0.	8.	2.	1865.	274.
SEP 77	1.	0.	6.	2.	1871.	275.
OCT 77	1.	0.	5.	1.	1876.	275.
NOV 77	1.	0.	4.	1.	1880.	276.
DEC 77	1.	0.	3.	1.	1883.	276.
JAN 78	0.	0.	2.	1.	1885.	277.


SPERRY UNIVAC PROG 3
CMD & CTL; REAL TIME
TDS DESIGN; 53% HOL
26,600 DSLOC; MEM - .5
PAGES OF DOC - 3507
MANLOADING PROFILE
MANPOWER (PEOPLE)
100
90
80
70
60
50
40
30
20
10
0
Jan Feb Mar Apr May June July Aug Sept Oct Nov Dec
19
26 Mos.
MM = 379

SPERRY UNIVAC PROG 3
SLIM MANLOADING PROFILE
FOR 399 MM OF DEVELOPMENT
EFFORT. 16,724 HOL EQUIV.
SOURCE STATEMENTS



Page intentionally left blank

Page intentionally left blank

PANEL #5

MODELS AND METRICS OF SOFTWARE DEVELOPMENT

B. Curtis, General Electric
J. Musa, Bell Labs
A. Stone, General Electric

PROGRAM COMPLEXITY AND SOFTWARE ERRORS: A FRONT END FOR RELIABILITY

Dr. Bill Curtis
Software Management Research
Information Systems Programs
General Electric Company
Arlington, Virginia

Error analysis and software complexity have received increased attention in software engineering research over the past several years. The study of software errors has been necessitated by the emphasis on software reliability. Models such as the one presented by John Musa in this volume statistically model such phenomena as the mean-time-between-failures or the probability of a failure within a given unit of time. As John indicates, one of the parameters required as input to this model is the number of errors existing in the software.

There are several ways to estimate the number of errors in a piece of software. One is the actuarial approach which assumes there are so many errors in a given number of lines of code. A number frequently passed about is one error per one hundred lines. This approach assumes that all software is created equal and ignores the advances that have been made during recent years in analyzing software characteristics. An alternative approach recognizes these gains in relating software characteristics to such factors as the error-proneness of a section of code or the difficulty which will be experienced in maintaining the code. The purpose of this paper is to review recent research on software complexity metrics to determine whether knowing something about software characteristics improves our ability to predict the number of errors it contains or the amount of effort required to maintain it.

If we can validate the use of software metrics for predicting the number of errors in software and the difficulty experienced in correcting them, then such metrics will prove a valuable addition to both quality assurance and management information systems. During the design phase, metric values can be estimated from relevant design information to predict problems which will be experienced during coding. Values computed on the actual code can be used in predicting testing results, number of delivered bugs, and ease of maintenance. Although a large number of metrics have been presented in the literature, two seem to have received the most attention in empirical research. I will focus on these two metrics in the remainder of this paper.

Thomas McCabe (1976) developed a complexity measure based on the cyclomatic number from graph theory. McCabe counts the number of regions in a graph of the control flow of a computer program. His metric represents the number of basic control path segments which when combined will generate every possible path through the program. Thus, McCabe has measured the complexity of the control structure. Schneidewind and Hoffmann (1979) demonstrated that the cyclomatic number and the reachability measure which can be computed from it were superior to the number of source statements in predicting the number of errors in a section of code and the time required to find and fix them. Feuer and Fowlkes (1979) also demonstrated that the node count was related to the time to repair errors. However, their data indicated that different prediction equations should be used with different types of errors. Separate prediction equations might be possible when we have (1) developed more robust error classification schemes, and (2) progressed past predicting gross errors to predicting types of errors.

Another approach to software complexity was presented by Maurice Halstead (1977) in his theory of Software Science. Halstead maintained that the amount of effort required to generate a program can be derived from simple counts of distinct operators and operands and the total frequencies of operators and operands. These quantities can be used to calculate the number of mental comparisons required to generate a program. Halstead's effort metric, E, expresses the complexity of computer software in psychological terms. Halstead also developed a metric to estimate the number of delivered errors in a system. This metric is based on the notion that there is a limited amount of code that a programmer can mentally grasp at a single time. When a section of code exceeds this value it is likely that the programmer made at least one mistake in producing it. Halstead predicts the number of errors by dividing the total volume of code by this critical level for error-prone code.

Bell and Sullivan (1974) presented a scatterplot which suggested that there was some validity to Halstead's notion of a critical value for error-free code. In their data no program with a Halstead volume above 260 was error-free, while only one program below this level had an error. Subsequently, both Cornell and Halstead (1976) and Fitzsimmons and Love (1978) found correlations of 0.75 and above between Halstead's metrics and the number of errors found in various software products. In a debugging study we recently completed at G.E. (Curtis, Milliman, and Sheppard, 1979) the Halstead and McCabe metrics were better predictors of the time required to find a bug than was lines of code.

In studying some error data provided us by Rome Air Development Center, Phil Milliman and I (1979) found Halstead's metric a remarkably accurate predictor of delivered bugs in a system developed with modern programming practices and tools. However, the prediction was poor in a system developed with conventional techniques. The types of errors experienced in the former system were typical when compared to the types of errors reported in other systems (in particular to several reported by TRW). Phil and I also observed that the error ratio reported during the final months of development was an excellent predictor of post-development test errors. The error ratio represents the number of failed runs divided by the total number of runs. We observed a linearly decreasing trend in the error ratio during the final 9 months of development. When we extrapolated this trend into post-development testing, we observed a good prediction of the number of errors detected.

We suspect from the data we have observed that the prediction of errors and maintenance resources will be more accurate on projects guided by modern programming practices. We believe that such practices will reduce the amount of variation in performance and quality resulting from such sources as individual differences among programmers, the programming environment, etc. That is, a structured discipline constrains the amount of variation in the way software is developed. Since this variation is a source of error in predictions, the ability to predict various software-related criteria (such as number of errors) should improve.

Based on the brief review of empirical research presented here, I propose the following conclusions, but agree that much more data is needed to substantiate them.

- Measures of software characteristics can be used to predict the number of errors in a portion of code and the effort required to find and correct them. Such measures will be more valuable than an actuarial approach based on lines of code.

- Different predictive plots may be observed for different classes of errors (computational, logic, interface, etc.)

- Metrics should be calculated at the appropriate level (subroutine, module, etc.) for explaining the results.

- The prediction of software reliability and of maintenance requirements can begin early in the software development cycle, and improvements can be made and monitored if feedback is provided for improving software quality.

ACKNOWLEDGEMENTS

I would like to thank Sylvia Sheppard and Elizabeth Kruesi for their comments, Beverly Day for manuscript preparation, and Lou Oliver for his support and encouragement. Work resulting in this paper was supported by the Office of Naval Research, Engineering Psychology Programs (Contract #N000014-79-C-0595) and the General Electric Company (IR&D Project 79D6A02). However, the opinions expressed in this paper are not necessarily those of the Department of the Navy or the General Electric Company.

REFERENCES

Bell, D. E. and J. E. Sullivan, Further investigations into the complexity of software (Tech. Rep. MTR-2874). Bedford, MA: MITRE, 1974.

Cornell, L. M. and M. H. Halstead, Predicting the number of bugs expected in a program module (Tech. Rep. CSD-TR-205). West Lafayette, IN: Purdue University, Computer Science Department, 1976.

Curtis, B. and P. Milliman, A matched project evaluation of modern programming practices (RADC-TR-79, 2 vols.). Griffiss AFB, NY: Rome Air Development Center, 1979.

Curtis, B., S. B. Sheppard, and P. Milliman, Third time charm: Stronger prediction of programmer performance by software complexity metrics. In Proceedings of the Fourth International Conference on Software Engineering. New York: IEEE, 1979.

Feuer, A. R. and E. B. Fowlkes, Some results from an empirical study of computer software. In Proceedings of the Fourth International Conference on Software Engineering, New York: IEEE, 1979.

Fitzsimmons, A. B. and L. T. Love, A review and evaluation of software science. ACM Computing Surveys, 1978, 10, 3-18.

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

McCabe, T. J., A complexity measure. IEEE Transactions on Software Engineering, 1976, 2, 308-320.

Schneidewind, N. F. and H. M. Hoffmann, An experiment in software error data collection and analysis. IEEE Transactions on Software Engineering, 1979, 5, 276-286.

GENERAL ELECTRIC COMPANY

SPACE DIVISION

NEEDS RELATING TO ERROR ANALYSIS

INFORMATION SYSTEMS PROGRAMS

SOFTWARE MANAGEMENT RESEARCH

NEEDS	USES
PREDICTORS OF THE NUMBER OF ERRORS RESIDENT IN A PORTION OF CODE	INPUTS INTO SOFTWARE RELIABILITY MODELS
PREDICTORS OF THE TIME REQUIRED TO FIND AND CORRECT SOFTWARE ERRORS	ESTIMATION OF TESTING AND MAINTENANCE RESOURCES

GENERAL ELECTRIC COMPANY

SPACE DIVISION

CANDIDATE PREDICTORS

INFORMATION SYSTEMS PROGRAMS

SOFTWARE MANAGEMENT RESEARCH

ACTUARIAL DATA

SOFTWARE CHARACTERISTICS

- CYCLOMATIC NUMBER
- SOFTWARE SCIENCE

DOES KNOWING SOMETHING ABOUT THE CHARACTERISTICS OF THE CODE IMPROVE OUR ABILITY TO PREDICT THE NUMBER OF ERRORS IT CONTAINS?

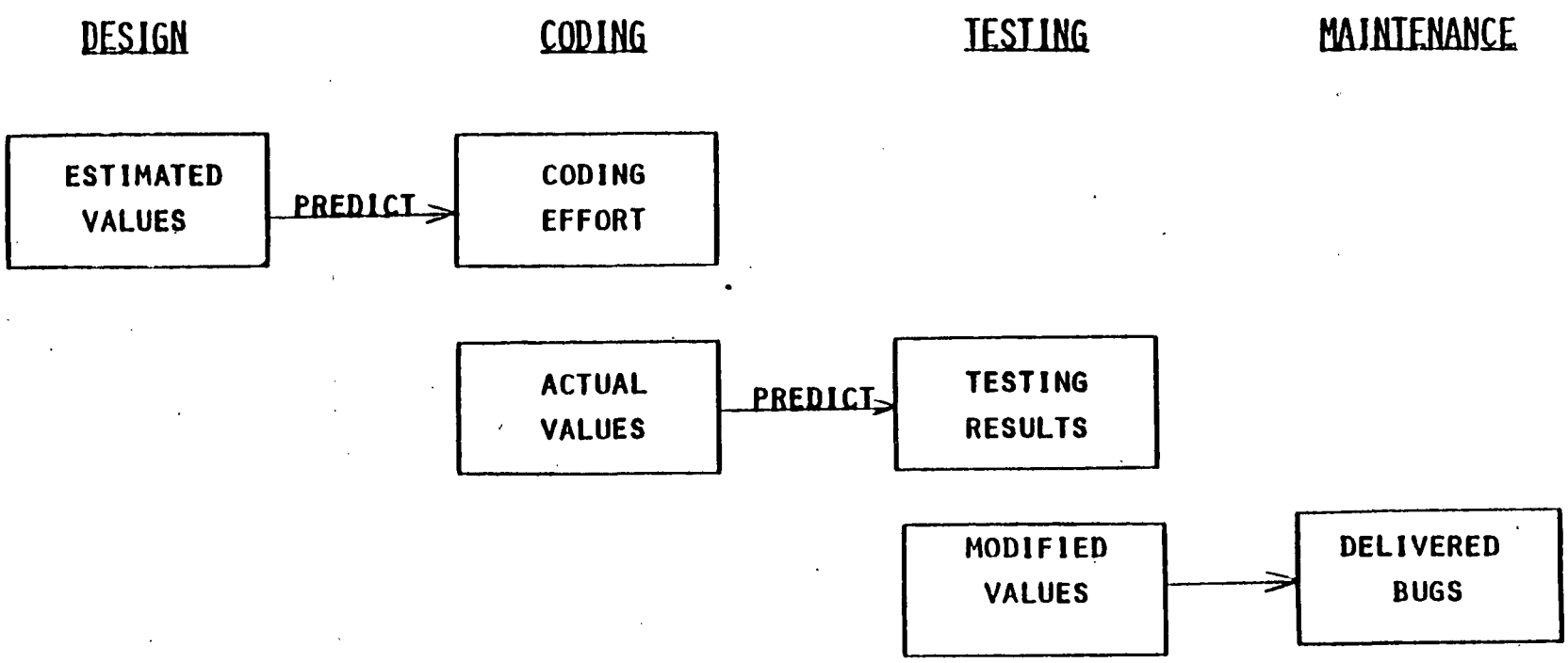
GENERAL ELECTRIC COMPANY

SPACE DIVISION

THE USE OF SOFTWARE METRICS IN A MANAGEMENT INFORMATION SYSTEM

INFORMATION SYSTEMS PROGRAMS

SOFTWARE MANAGEMENT RESEARCH



GENERAL ELECTRIC COMPANY

SPACE DIVISION

THOMAS J. McCABE
A COMPLEXITY MEASURE (1976)

INFORMATION SYSTEMS PROGRAMS

SOFTWARE MANAGEMENT RESEARCH

EQUATION:

$v(G) = \#\ \text{EDGES} - \#\ \text{NODES} + 2(\#\ \text{CONNECTED COMPONENTS})$

OR

$v(G) = \#\ \text{PREDICATE NODES} + 1$

OR

$v(G) = \#$ REGIONS IN A PLANAR GRAPH OF THE CONTROL FLOW.

DESCRIPTION:

McCABE'S METRIC REPRESENTS THE NUMBER OF LINEARLY INDEPENDENT CONTROL PATHS COMPRISING A PROGRAM. THAT IS, THE NUMBER OF BASIC CONTROL PATH SEGMENTS WHICH WHEN COMBINED WILL GENERATE EVERY POSSIBLE PATH THROUGH THE PROGRAM. McCABE'S v(G) REPRESENTS A MEASURE OF COMPUTATIONAL COMPLEXITY.

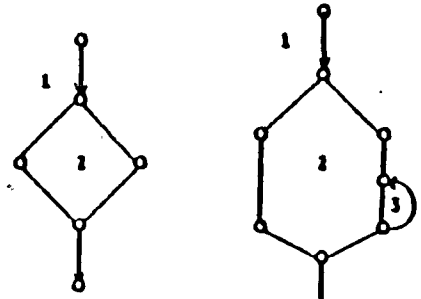
GENERAL ELECTRIC COMPANY

SPACE DIVISION

COMPUTATION OF McCABE'S v(G)

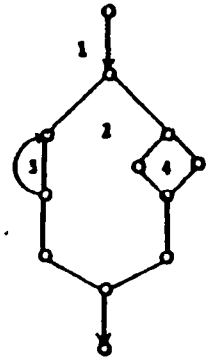
INFORMATION SYSTEMS PROGRAMS

SOFTWARE MANAGEMENT RESEARCH

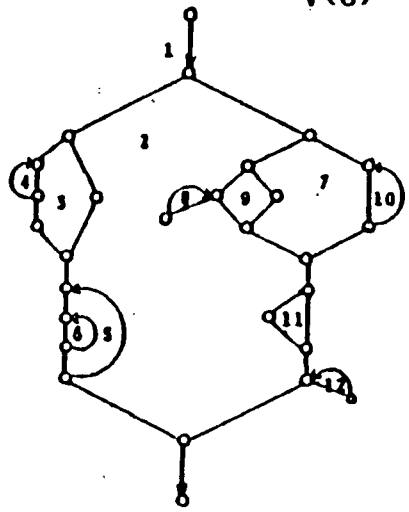


v(G) = 2

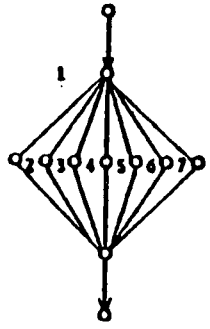
v(G) = 3



v(G) = 4



v(G) = 12



v(G) = 7

GENERAL ELECTRIC COMPANY

SPACE DIVISION

SCHNEIDEWIND AND HOFFMANN'S DATA (1979)

INFORMATION SYSTEMS PROGRAMS

SOFTWARE MANAGEMENT RESEARCH

PREDICTOR	NUMBER OF PROCEDURES	CORRELATIONS		
		# OF ERRORS	FIND TIME	FIX TIME
CYCLOMATIC NUMBER	31	.78	.67	.72
REACHABILITY	20	.77	.90	.66
SOURCE STATEMENTS	20	.59	.59	.51

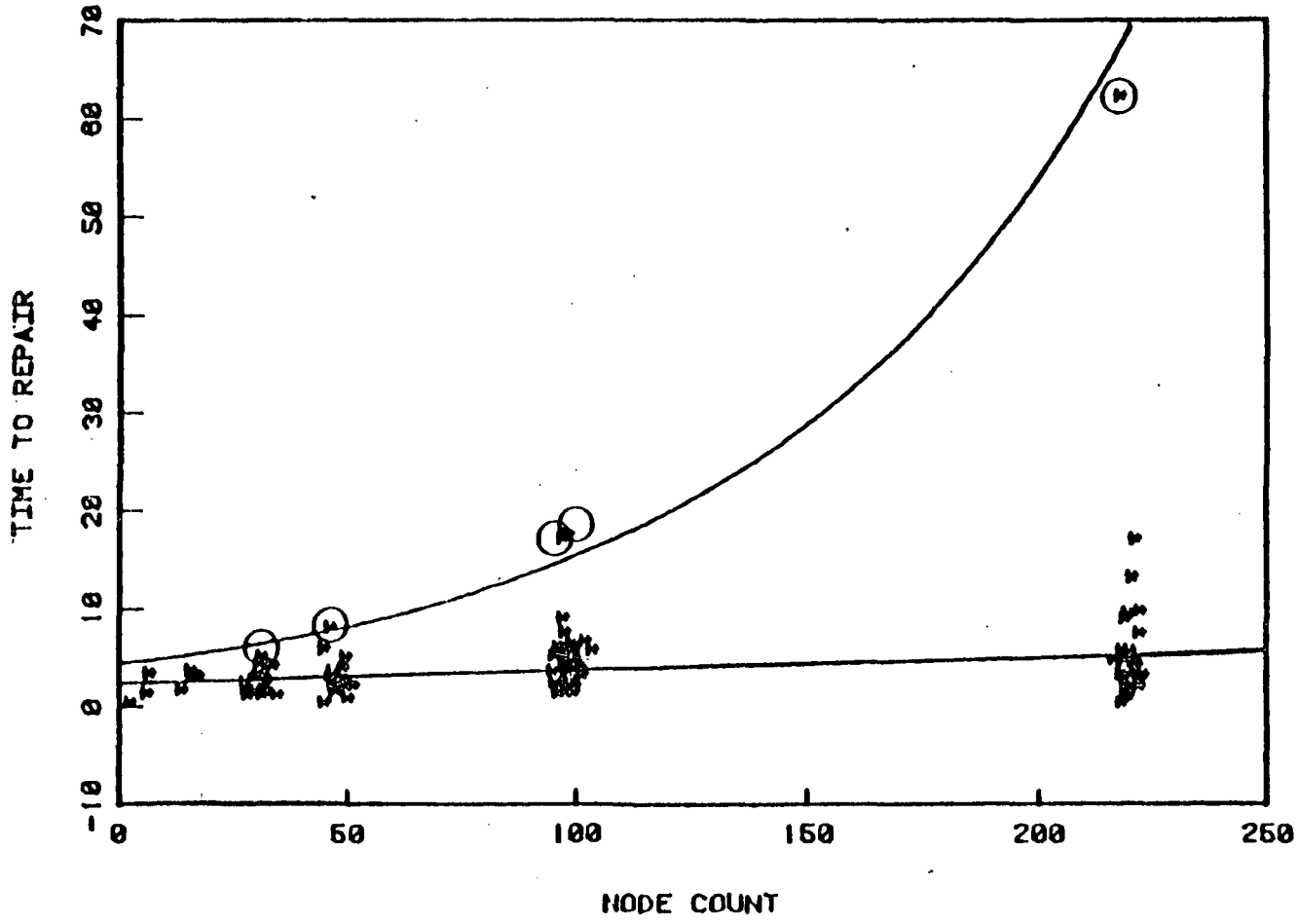
GENERAL ELECTRIC COMPANY

SPACE DIVISION

FEUER AND FOWLKES DATA (1979)

INFORMATION SYSTEMS PROGRAMS

SOFTWARE MANAGEMENT RESEARCH



GENERAL ELECTRIC COMPANY

SPACE DIVISION

MAURICE H. HALSTEAD

ELEMENTS OF SOFTWARE SCIENCE (1977)

INFORMATION SYSTEMS PROGRAMS

SOFTWARE MANAGEMENT RESEARCH

EQUATION:

$$E = \frac{\eta_1 N_2 (N_1 + N_2) \text{Log}_2 (\eta_1 + \eta_2)}{2\eta_2}$$

WHERE,

η_1 = # OF UNIQUE OPERATORS

η_2 = # OF UNIQUE OPERANDS

N_1 = F OF OPERATORS

N_2 = F OF OPERANDS

DESCRIPTION:

THE AMOUNT OF EFFORT REQUIRED TO GENERATE A PROGRAM CAN BE DERIVED FROM SIMPLE COUNTS OF DISTINCT OPERATORS AND OPERANDS AND THE TOTAL FREQUENCIES OF OPERATORS AND OPERANDS. THESE QUANTITIES CAN BE USED TO CALCULATE THE NUMBER OF MENTAL COMPARISONS REQUIRED TO GENERATE A PROGRAM. HALSTEAD'S EFFORT METRIC, E, EXPRESSES THE COMPLEXITY OF COMPUTER SOFTWARE IN PSYCHOLOGICAL TERMS.

GENERAL ELECTRIC COMPANY

SPACE DIVISION

HALSTEAD'S MEASURE OF DELIVERED BUGS

INFORMATION SYSTEMS PROGRAMS

SOFTWARE MANAGEMENT RESEARCH

$$B = V/E_{CRIT}$$

$$= \frac{V_\lambda}{13,824}$$

WHERE,

V = VOLUME

E_{CRIT} = THE MEAN NUMBER OF ELEMENTARY DISCRIMINATIONS BETWEEN POTENTIAL ERRORS IN PROGRAMMING

λ = LEVEL OF THE IMPLEMENTATION LANGUAGE

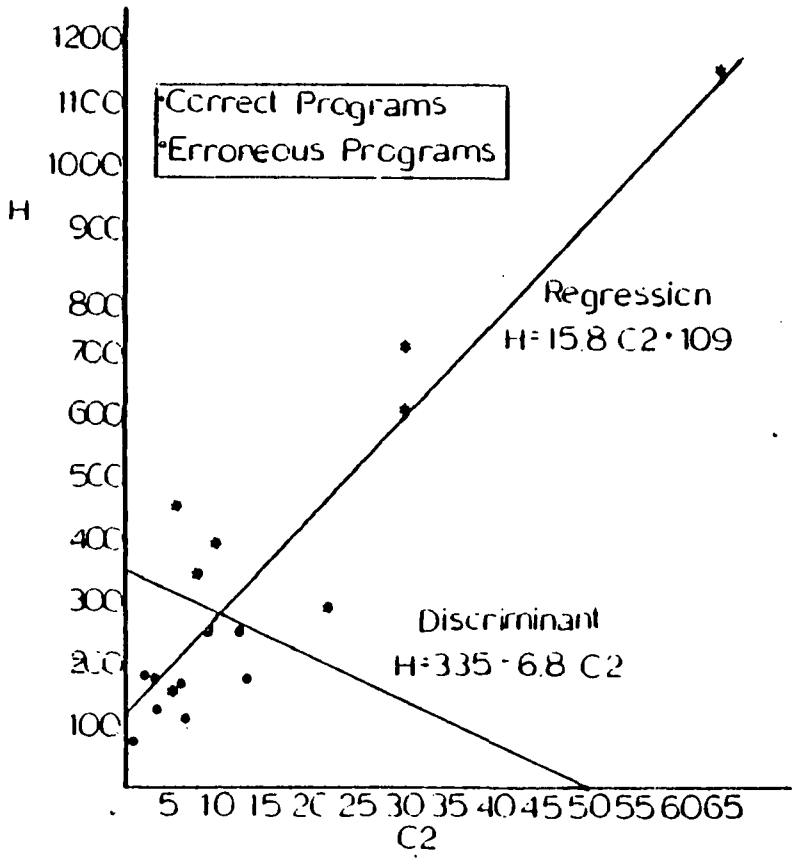
GENERAL ELECTRIC COMPANY

SPACE DIVISION

BELL AND SULLIVAN'S DATA (1974)

INFORMATION SYSTEMS PROGRAMS

SOFTWARE MANAGEMENT RESEARCH



GENERAL ELECTRIC COMPANY

SPACE DIVISION

CORNELL AND HALSTEAD'S DATA (1976)

INFORMATION SYSTEMS PROGRAMS

SOFTWARE MANAGEMENT RESEARCH

MILLIONS OF MENTAL DISCRIMINATIONS	NUMBER OF ERRORS	
	ACTUAL	PREDICTED
170.3	102	102
15.3	18	20
322.6	146	156
28.2	26	30
100.2	71	71
65.5	37	54
6.5	16	11
58.5	50	50
135.9	80	88
903.0	546	582

R = .99

GENERAL ELECTRIC COMPANY

SPACE DIVISION

FITZSIMMONS AND LOVE'S DATA (1978)

INFORMATION SYSTEMS PROGRAMS

SOFTWARE MANAGEMENT RESEARCH

SUBSYSTEM	NUMBER OF MODULES	RANGE OF STMTS PER MODULE	TOTAL STMTS	CORRELATION OF E WITH ERRORS
COMMAND EXECUTIVE	47	70-7100	53,920	.81
DATABASE MANAGER	42	10-6050	64,910	.75
REPORT GENERATOR	51	50-3700	47,450	.75
TOTAL	140	10-7100	166,280	.77

GENERAL ELECTRIC COMPANY

SPACE DIVISION

CURTIS, SHEPPARD, & MILLIMAN'S DATA (1979)

INFORMATION SYSTEMS PROGRAMS

SOFTWARE MANAGEMENT RESEARCH

	E	V(G)	LENGTH
INTERRELATIONSHIPS			
V(G)	.76***		
LENGTH	.56***	.90***	
TIME TO FIND BUG:			
TOTAL PROGRAM	.75***	.65***	.52***
SUBROUTINE	.66***	.63***	.67***

NOTE: N = 27

*** $P \leq .001$

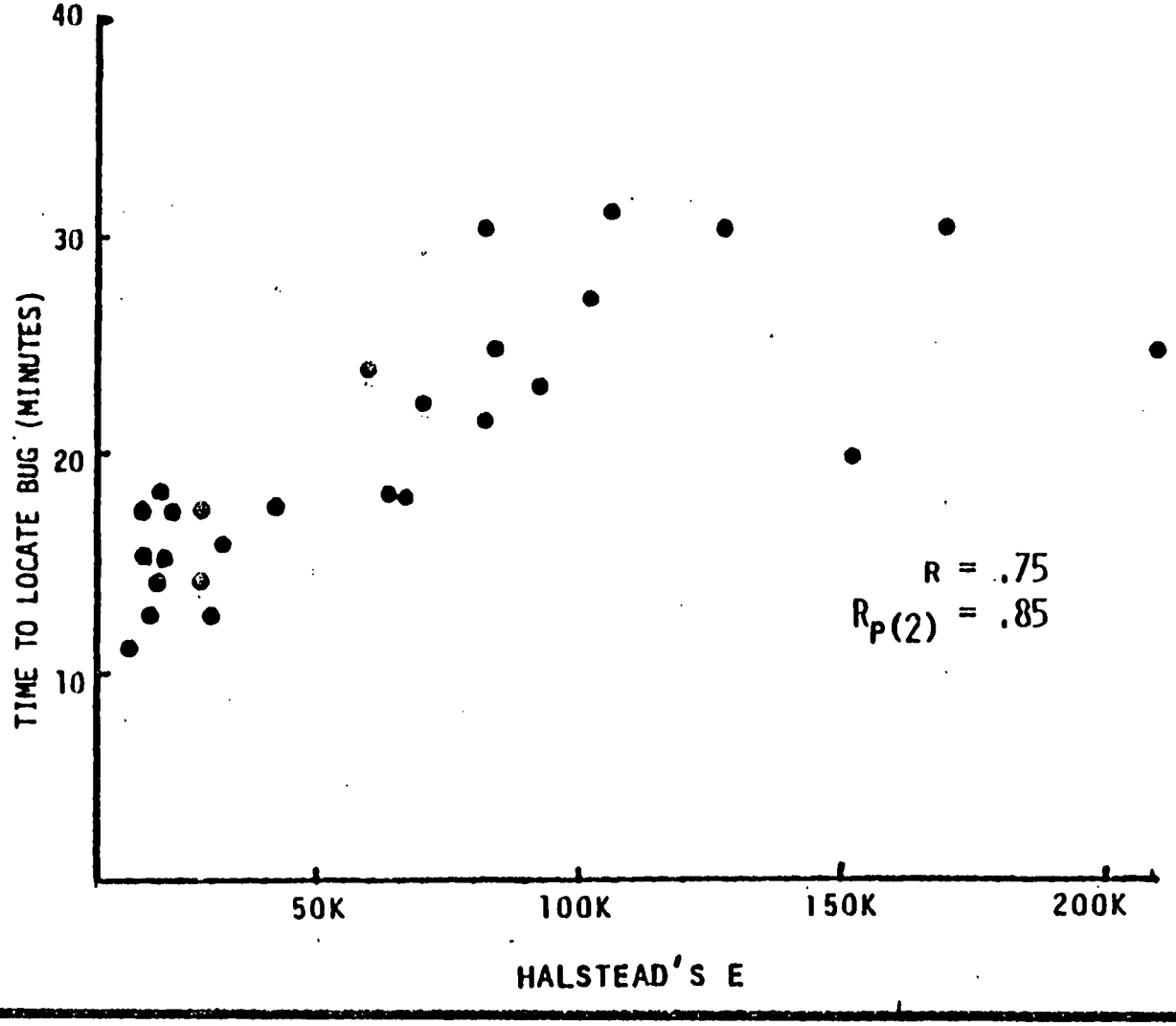
GENERAL ELECTRIC COMPANY

SPACE DIVISION

SCATTERPLOT OF HALSTEAD'S E WITH DEBUGGING TIME

INFORMATION SYSTEMS PROGRAMS

SOFTWARE MANAGEMENT RESEARCH



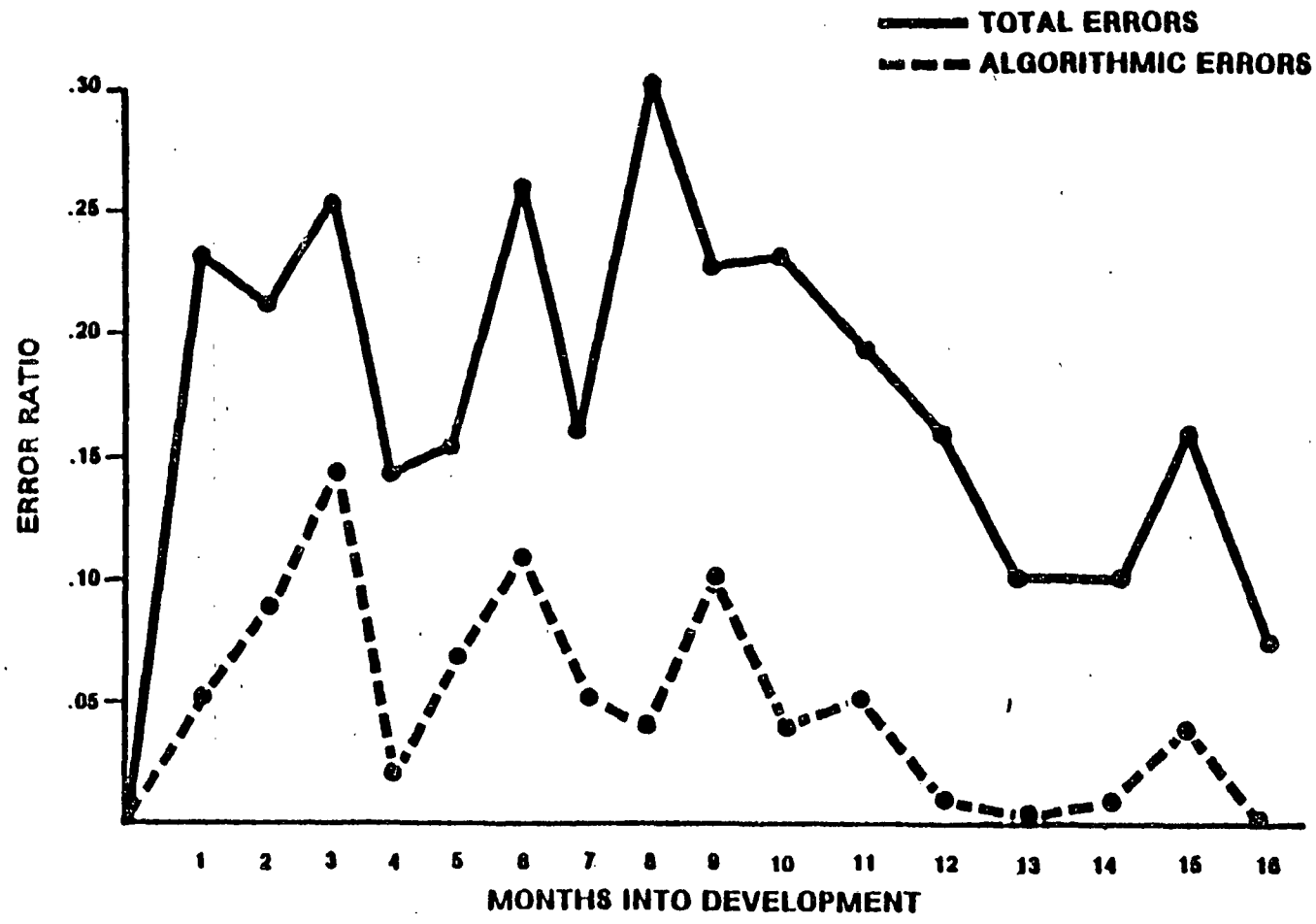
GENERAL ELECTRIC COMPANY

SPACE DIVISION

CURTIS AND MILLIMAN'S DATA (1979)
ERROR RATIO BY MONTH

INFORMATION SYSTEMS PROGRAMS

SOFTWARE MANAGEMENT RESEARCH



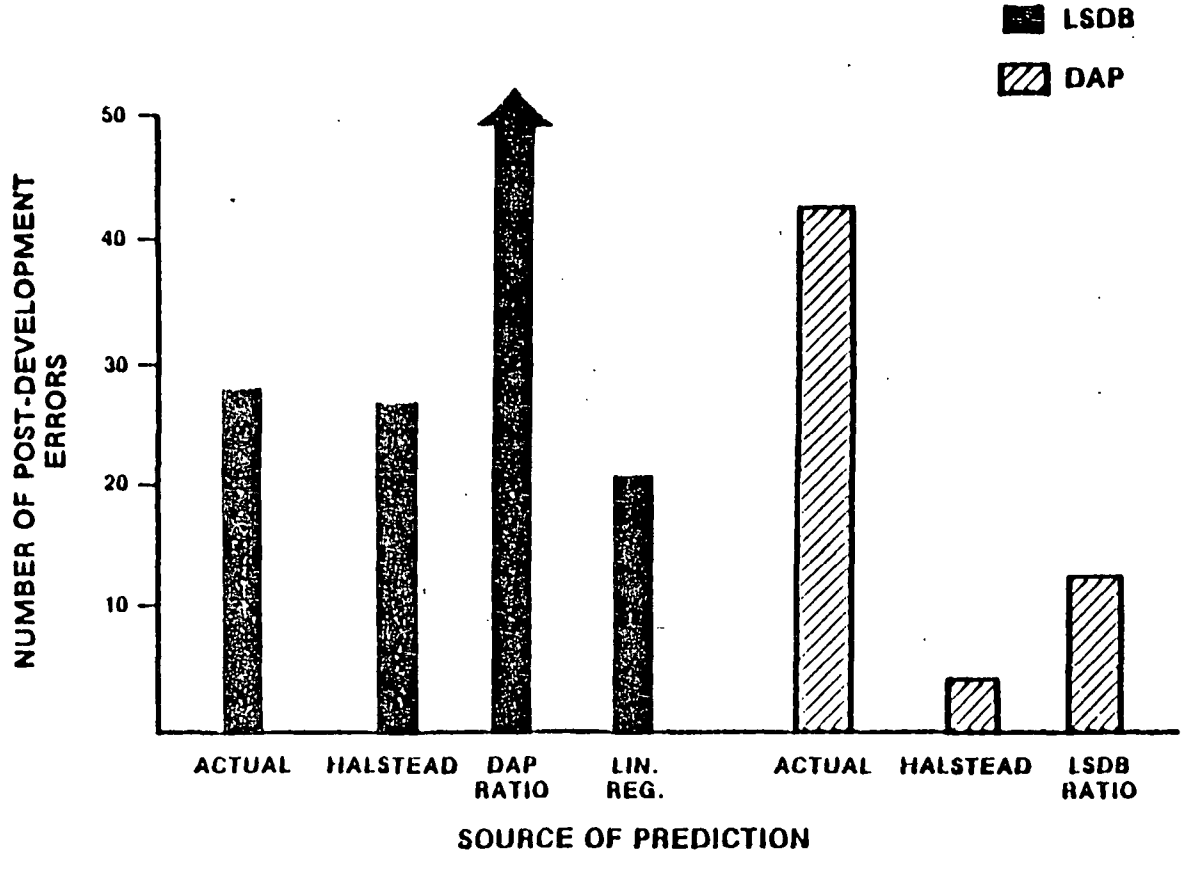
GENERAL ELECTRIC COMPANY

SPACE DIVISION

PREDICTION OF POST-DEVELOPMENT ERRORS

INFORMATION SYSTEMS PROGRAMS

SOFTWARE MANAGEMENT RESEARCH



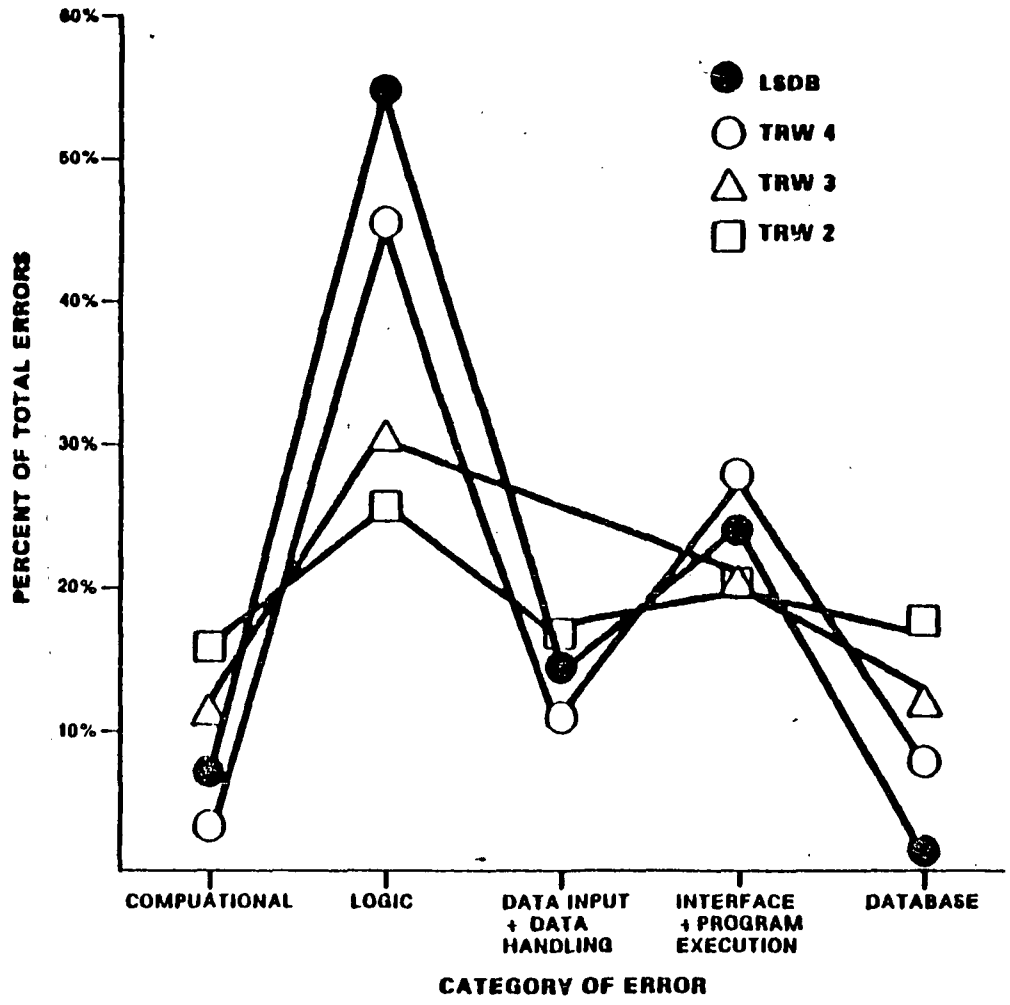
GENERAL ELECTRIC COMPANY

SPACE DIVISION

COMPARISON OF ERROR DISTRIBUTIONS

INFORMATION SYSTEMS PROGRAMS

SOFTWARE MANAGEMENT RESEARCH



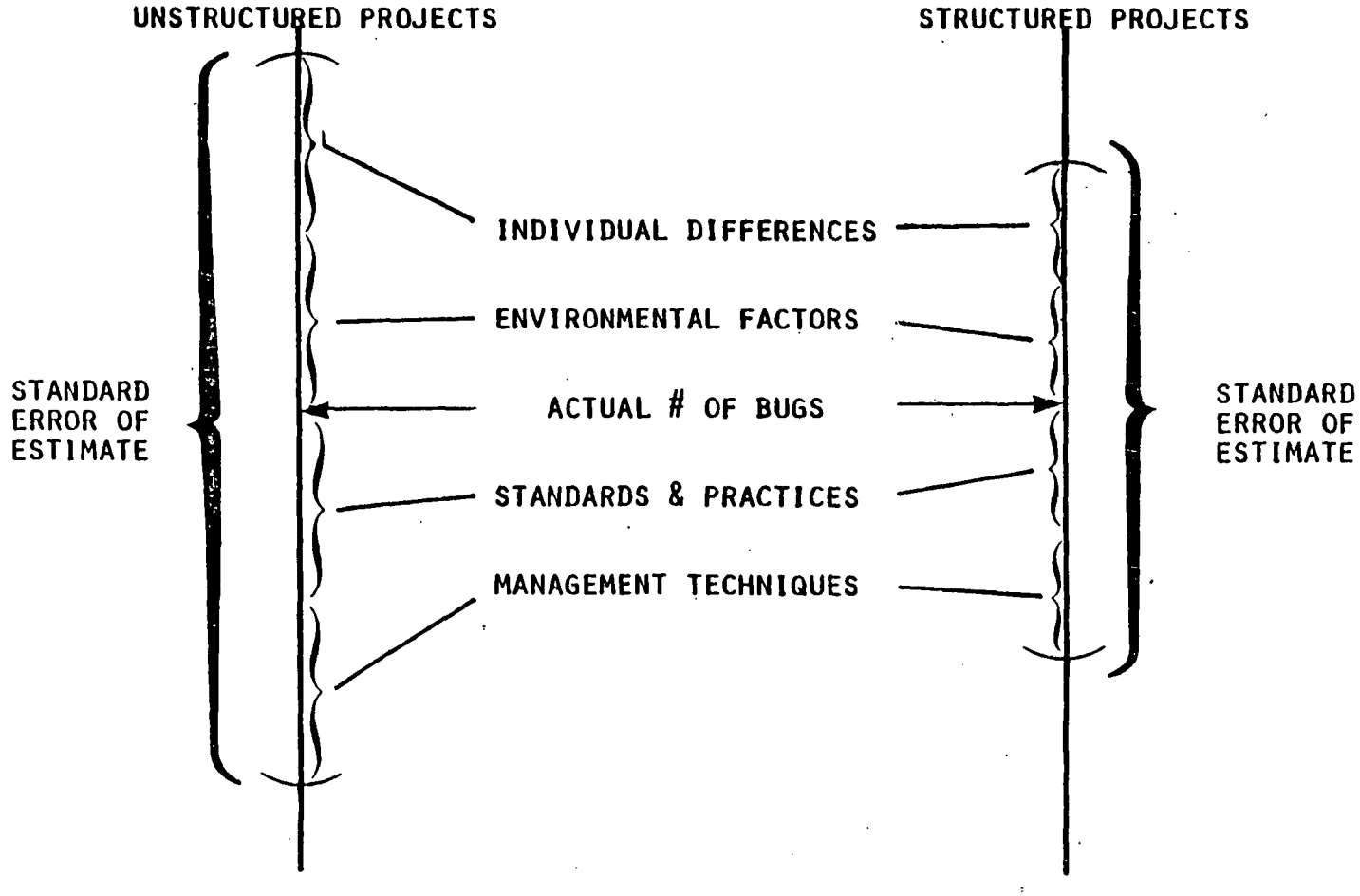
GENERAL ELECTRIC COMPANY

SPACE DIVISION

FACTORS INFLUENCING THE ACCURACY OF PREDICTION

INFORMATION SYSTEMS PROGRAMS

SOFTWARE MANAGEMENT RESEARCH



GENERAL ELECTRIC COMPANY

SPACE DIVISION

CONCLUSIONS

INFORMATION SYSTEMS PROGRAMS

SOFTWARE MANAGEMENT RESEARCH

- MEASURES OF SOFTWARE CHARACTERISTICS CAN BE USED TO PREDICT THE NUMBER OF ERRORS IN A PORTION OF CODE AND THE EFFORT REQUIRED TO FIND AND CORRECT THEM
- DIFFERENT PREDICTIVE PLOTS WILL BE OBSERVED FOR DIFFERENT CLASSES OF ERRORS
- THERE ARE OPTIMAL LEVELS IN THE CODE FOR CALCULATING METRICS
- THE PREDICTION OF SOFTWARE RELIABILITY AND MAINTENANCE REQUIREMENTS CAN BEGIN EARLY IN THE SOFTWARE DEVELOPMENT CYCLE, AND IMPROVEMENTS CAN BE MONITORED

SOFTWARE RELIABILITY MODELING – WHERE ARE WE AND WHERE SHOULD WE BE GOING?

THE NEED FOR SOFTWARE RELIABILITY MODELING

It may be argued that software reliability metrics are needed, most importantly, because no field can really mature until it can be described in a quantitative fashion. However, there are also some very specific reasons for a quantitative approach to software reliability. One needs software liability figures in order to do a good job of system engineering: to examine the trade offs between reliability and cost and reliability and schedules, to determine what reliability figure optimizes overall life cycle costs, to plan allocation of resources, and to specify reliability to a contractor who is developing software for you. Another large area of application is project management, where software reliability measures are needed for progress monitoring, scheduling and investigation of managerial alternatives. The length of a test period and hence the overall length of a project is highly correlated with the reliability requirements for the project. Therefore, reliabilities are intimately tied up with schedules. Changes in resources available to the project affect both reliability and schedules and one can be exchanged for the other. Reliability metrics offer an excellent means of evaluating the performance of operational software and controlling changes to it. Since change usually involves a degradation of reliability, one may use reliability performance objectives as a means for determining when software changes can be allowed and perhaps even how large they can be. Finally, reliability is one of the important parameters that should be used in investigating the benefits (or lack of benefits) of proposed new software engineering technology.

SOFTWARE RELIABILITY FUNCTIONS

Hecht [1] has categorized software reliability functions into measurement, estimation and prediction. This classification is used in this paper with some modification and extension. Software reliability is defined as the probability that a program will execute without failure caused by software for a specified time in a specified environment. The term "failure" refers to an unacceptable departure from proper operation. The term "unacceptable" must be defined by the customer. The "measurement" of software reliability is based on failure interval data obtained by running the program in its actual operating environment. Software reliability "estimation" refers to the process of determining software reliability metrics based on operation in a test environment. It should be noted that estimation can be performed with respect to present or future reliability quantities. The term software reliability "prediction" refers to the process of computing software reliability quantities from program data which does not include failure intervals. Typically, software reliability prediction takes into account factors such as size and complexity of the program, and is normally performed during a program phase prior to test. Note that future estimation might be thought of by some as prediction; we are deliberately making a careful distinction in terminology.

The various applications of software reliability metrics are closely tied to the three functions that have just been defined. System engineering primarily relies upon prediction; project management, upon estimation; and operational software management and evaluation of software engineering technology, upon measurement.

J. Musa
Bell Lab.
1 of 11

SOFTWARE RELIABILITY MODELS

Most of the work that has been done in the field of software reliability falls in one of six categories: calendar time models, the execution time model, Bayesian models, semi-Markov models, deterministic models and input space approaches. The initial approach to software reliability was through calendar time models; that is, attempts were made to look of reliability phenomena such as failures, reliability, mean-time-to-failure (MTTF), etc. as functions of calendar time. These early models focused attention on the problem of software reliability and contributed many valuable concepts toward the further development of the theory. [2-5]

However, the failure-inducing stress placed on software is related closely to execution time (CPU time) and not calendar time. The execution time model [6-11] recognizes this fact. It has been extensively tested on more than 20 software systems and the validity of the assumptions made in deriving the model has been carefully examined. [12]

Littlewood and Verrall [13] have proposed a Bayesian model that is perhaps the most mathematically elegant of the software reliability models, but it is, unfortunately, difficult to understand, and computations based on it are lengthy and costly. A model that focuses specifically on the problem of imperfect fault correction has been developed by Goel and Okumoto [14]; it is based on a view of fault correction as a semi-Markov process. The concept of imperfect fault correction is incorporated in the execution time model in a simpler fashion. Deterministic models have been proposed [15, 16] but they have not been validated.

It would appear that deterministic models oversimplify the failure detection and correction process and are not efficient in using the information available to them. Bayesian models perhaps represent the other extreme, in that both failure intervals and failure process parameters are viewed as being random. The execution time model takes the intermediate approach of considering failure intervals random but failure process parameters as varying with execution time in a deterministic fashion.

A final viewpoint, the input space approach, is based on enumerating all the possible sets of input or environmental conditions for a program and determining the proportion of these that result in successful operation. Although this approach is theoretically appealing, the large number of possible input sets for any useful program makes it impractical. The counts would have to be weighed by run times and frequencies of operation for the various input sets, in order to provide results that would be compatible with hardware reliability theory.

EXECUTION TIME MODEL

The execution time model permits the development of relationships that indicate number of failures experienced and present MTFF as functions of execution time (see Figures 1 and 2). It relates total failures and initial MTFF to the number of faults in the system. An initial estimate of the number of faults, prior to testing, can be determined from the size (and perhaps complexity) of the program. A debugging process model is provided which relates execution time and calendar time and thus allows execution time quantities to be converted into dates. The model can be used to make predictions of the remaining number of failures to be experienced, the execution time and the calendar time required to reach a MTTF objective. If this objective is set as the

criterion for terminating the project, completion dates can be predicted. As testing proceeds, two of the key parameters of the model can be statistically reestimated from failure intervals experienced. This permits the estimation of a number of derived quantities such as present MTTF and estimated completion date. The estimates made are maximum likelihood estimates; confidence intervals are also calculated.

Most of the assumptions that were made in deriving the execution time model have been validated [12] and experience has been gained with the model on a wide variety of software systems (more than 20 as of this date). A program is available [17, 18] to handle the statistical calculations. Sample output from the program is shown in Figure 3.

User comments indicate that the execution time model provides a good conceptual framework for viewing the software failure process. It is simple, its parameters are closely related to the physical world and it is compatible with hardware reliability theory. Most users feel that the benefits currently exceed the costs, which are basically data collection and computation. There have been two interesting side benefits. The process of defining just what constitutes a failure and the process of setting a MTTF objective have both been salutary in opening up communication between customer and developer.

STATE OF THE ART AND RESEARCH NEEDS

Software measurement can presently be achieved with excellent accuracy. Figure 4 illustrates a software system in the operational state. The maximum likelihood estimate and 75% confidence bounds are indicated for present MTTF. Variations in MTTF and the size of the confidence interval are generally highly correlated with periods of fault correction or the addition of new capabilities.

The quality of software reliability estimation is dependent upon the representativeness of testing; hence good test planning is esesntial. If one desires to know the absolute value of the MTTF, knowledge of the test compression factor is necessary. The test compression factor relates the amount of time spent in test with the equivalent amount of operating time represented. It is known theoretically how to compute this number but the only practical approach at present is to estimate it from a similar porject in a similar test environment. Research activity in this area would definitely be beneficial. One might characterize the present quality of software reliability estimation as good for present estimation and fair for future estimation. Future estimation also requires, in addition to the factors previously listed, a number of resource parameters. Data collection to determine the values of these parameters and the extent to which they vary between different projects or different classes of projects is urgently needed. Figure 5 illustrates the variation in present MTTF as the system test phase of a project proceeded (maximum likelihood estimate and 75% confidence bounds are indicated). Although the accuracy of the absolute estimates is dependent on the test compression factor, the relative values (i.e., denoting progress) are highly accurate.

The function of software reliability prediction needs the most work. However, it also offers great promise in terms of ultimate potential benefits [9]. All of the input quantities required for software estimation are needed for this function as well. In addition, one requires the number of faults inherent in the software, the fault exposure ratio, the fault reduction factor and the linear execution frequency. Figure 6 indicates the quantities and relationships involved in software

reliability perdiction. The number of faults inherent in the software N_0 must be determined from estimates of program size and data on fault densities. Data on fault densities is just beginning to accumulate but much more is needed, along with information on the variation of the fault density with program complexity and other factors. The fault reduction factor B indicates the ratio of net faults repaired to failures detected. It is a function of the test or operational environment and appears to be constant across similar environments. The initial MTTF, T_0, must be predicted from total failures M_0, from the linear execution frequency of the program f (throughput divided by object program size) and the fault exposure ratio K. The fault exposure ratio is expected to be dependent on the dynamic structure of the program and the degree to which faults are data dependent. Further investigation of the properties of this ratio and the factors upon which they depend is very important if we are to obtain good absolute software reliability predictions. Relative predictions can be made without this knowledge in many cases and they may be useful for many system engineering studies.

CONCLUSIONS

Software reliability has come a long way since its early beginnings in 1972. Many of the early problems have been solved and a reasonable amount of actual failure data has been collected. It may be seen from this paper that a number of problems remain to be solved and that new problems will probably suggest themselves as the field progresses. It is important, however, that we build upon the results that have already been achieved so as to maximize the efficiency of our efforts.

REFERENCES

1. H. Hecht, "Measurement, estimation, and prediction of software reliability," In Software Engineering Technology – Volume 2, Infotech International, Maidenhead, Berkshire, England, 1977, pp. 209–224; also in NASA Report CR145135, 1977 Jan.

2. Z. Jelinski and P. B. Moranda, "Software reliability research," in Statistical Computer Performance Evaluation, W. Freiberger, Ed. New York: Academic, 1972, pp. 465–484.

3. M. Shooman, "Probabilistic models for software reliability prediction," in Statistical Computer Performance Evaluation, see [2], pp. 485–502; also in 1972 Int: Symp. Fault-Tolerant Computing, Newton, Mass., 1972, June 21, pp. 211-215.

4. N. F. Schneidewind, "An approach to software reliability prediction and quality control," in 1972 Fall Joint Comput. Conf., AFIPS Conf. Proc., Vol. 41, Montvale, NJ: AFIPS Press, pp. 837–847.

5. G. J. Schick and R. W. Wolverton, "Assessment of software reliability," presented at 11th Annual Meeting of German Operations Research Society, Hamburg, Germany, 1972 Sep 6–8.

6. J. D. Musa, "A software reliability model," presented at NASA Software Engineering Workshop, Goddard Space Flight Center, Greenbelt, Maryland, 1977 Sep. 19.

7. J. D. Musa, "A theory of software reliability and its application," IEEE Trans. Software Engineering, Vol. SE-1, 1975 Sep., pp. 312-327.

8. J. D. Musa, "Software reliability measurement," in Software Phenomenology: Working Papers of the Software Life Cycle Management Workshop, Airlie, Va., 1977, Aug. 21-23, pp. 427-451. Also to be published in Journal of Systems and Software.

9. J. D. Musa, "Software reliability measures applied to system engineering," in 1979 NCC Proceedings, New York, N.Y., 1979 June 4-7, pp. 941-946.

10. J. D. Musa, "The use of software reliability measures in project management," in Proc. COMPSAC 78, Chicago, Ill., 1978 Nov. 14-16, pp. 493-498.

11. Patricia A. Hamilton and John D. Musa, "Measuring reliability of computation center software," in Proc. 3rd. Int. Conf. Soft. Eng., Atlanta, Ga., 1978 May 10-12, pp. 29-36.

12. J. D. Musa, "Validity of the execution time theory of software reliability," in IEEE Transactions on Reliability, Vol. R-28, No. 3, 1979 Aug., pp. 181-191.

13. B. Littlewood and J. L. Verrall, "A Bayesian reliability growth model for computer software," in 1973 IEEE Symp. Computer Software Reliability, New York, NY, 1973, Apr. 30-May 2, pp. 70-77.

14. A. L. Goel and K. Okumoto, Bayesian Software Prediction Models – An Imperfect Debugging Model for Reliability and Other Quantitative Measures of Software Systems, Rome Air Development Center Report RADC-TR-78-155, Vol. I.

15. H. Remus and S. Zilles, "Prediction and management of program quality," in Proc. 4th Int. Conf. on Software Engineering, Munich, Germany, 1979 Sep. 17-19, pp. 341-350.

16. I. Nathan, "A deterministic model to predict 'error free' status of complex software in development," in Proc. Workshop on Quantitative Software Models, Kiamesha Lake, N.Y., 1979 Oct. 9-11, to be published.

17. J. D. Musa, Program for software reliability and system test schedule estimation – user's guide, IEEE Computer Society Repository, Ref. No. R77-244.

18. J. D. Musa and P. A. Hamilton, Program for software reliability and system test schedule estimation – program documentation, IEEE Computer Society Repository, Ref. No. R277-243.

FAILURES EXPERIENCED VS. EXECUTION TIME

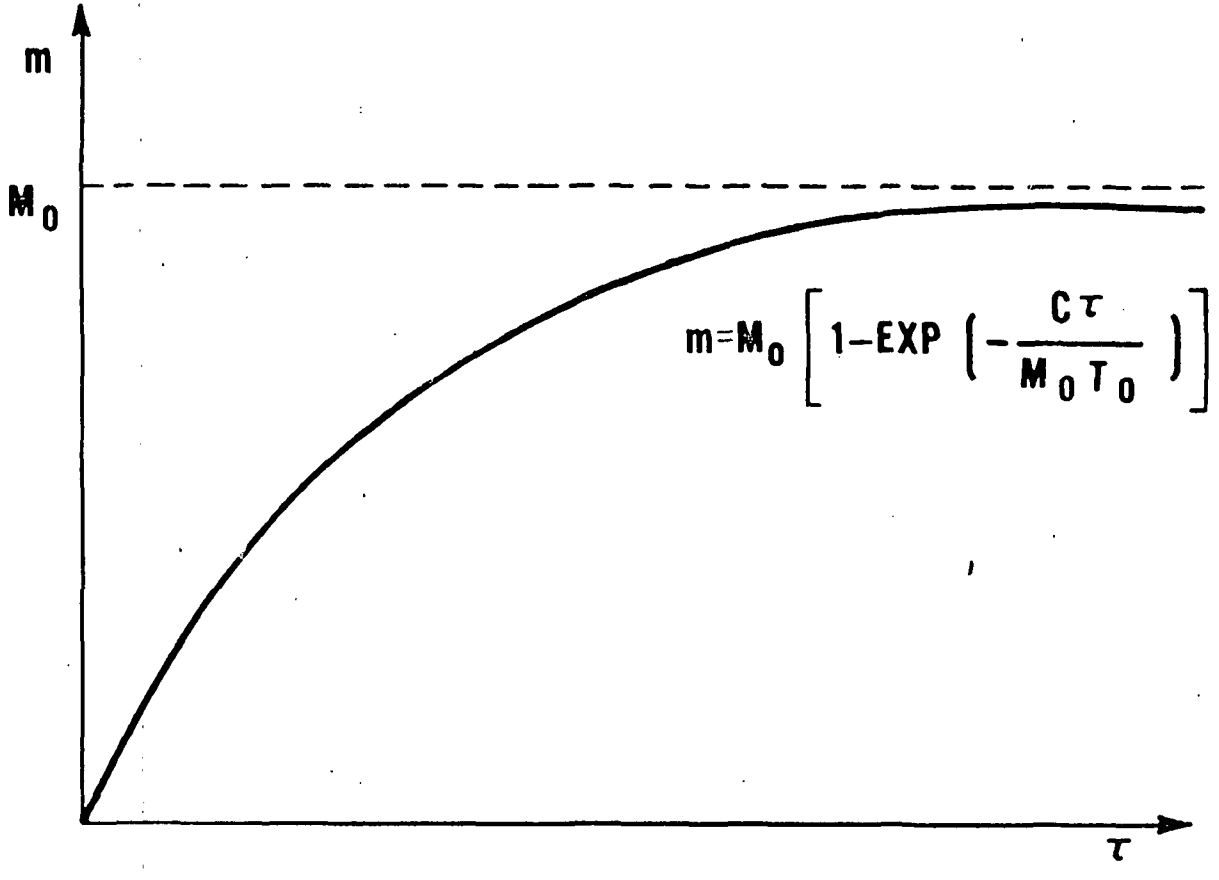


FIGURE 1. EXECUTION TIME MODEL RELATIONSHIP.

J. Musa
Bell Lab.
6 of 11

PRESENT MTTF VS. EXECUTION TIME

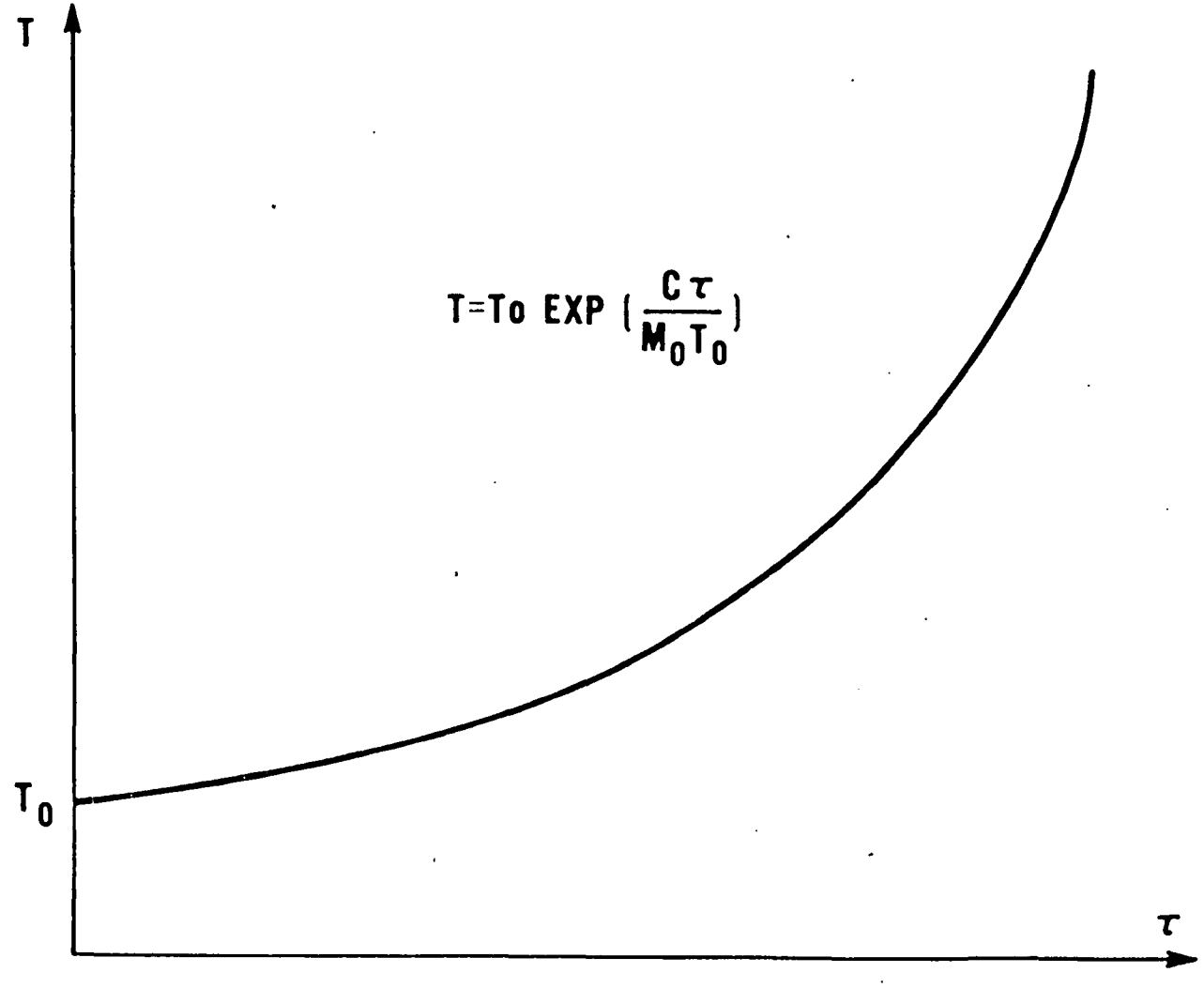


FIGURE 2. EXECUTION TIME MODEL RELATIONSHIP.

J. Musa
Bell Lab.
7 of 11

SOFTWARE RELIABILITY PREDICTION
PROJECT 1

BASED ON SAMPLE OF 136 TEST FAILURES
EXECUTION TIME IS 25.34 HRS
MTTF OBJECTIVE IS 27.80 HOURS
CALENDAR TIME TO DATE IS 96 DAYS
PRESENT DATE:11/ 9/73

	CONF. LIMITS				MOST		CONF. LIMITS		
	95%	90%	75%	50%	LIKELY	50%	75%	90%	95%
TOTAL FAILURES	136	136	136	138	142	148	152	163	182
INITIAL MTTF(HR)	0.522	0.617	0.701	0.744	0.847	0.949	0.992	1.08	1.17
PRESENT MTTF(HR)	999999	999999	999999	30.9	20.4	14.5	12.5	9.53	7.05
PERCENT OF OBJ	100.0	100.0	100.0	100.0	73.4	52.0	45.1	34.3	25.4
*** ADDITIONAL REQUIREMENTS TO MEET MTTF OBJECTIVE ***									
FAILURES	0	0	0	0	2	5	7	12	23
EXEC. TIME(HR)	0	0	0	0	2.46	6.09	7.94	12.4	19.4
CAL. TIME(DAYS)	0	0	0	0	0.958	2.85	4.03	7.39	13.8
COMPLETION DATE READY	110973	110973	110973	110973	111273	111473	111673	112173	112973

Figure 3. Sample Output from Software Reliability Measurement/ Estimation Program for Execution Time Model

SOFTWARE SYSTEM 4

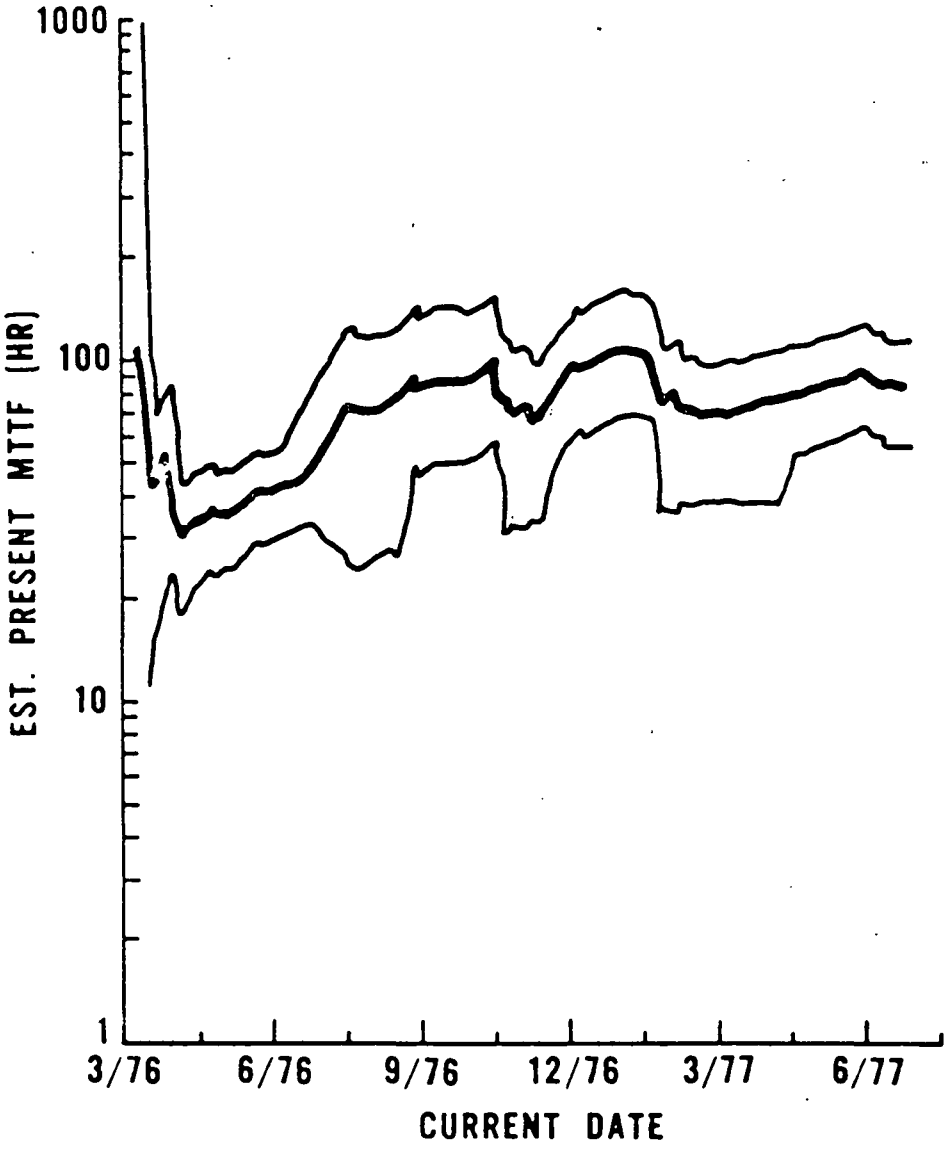


Figure 4. Software Reliability Measurement

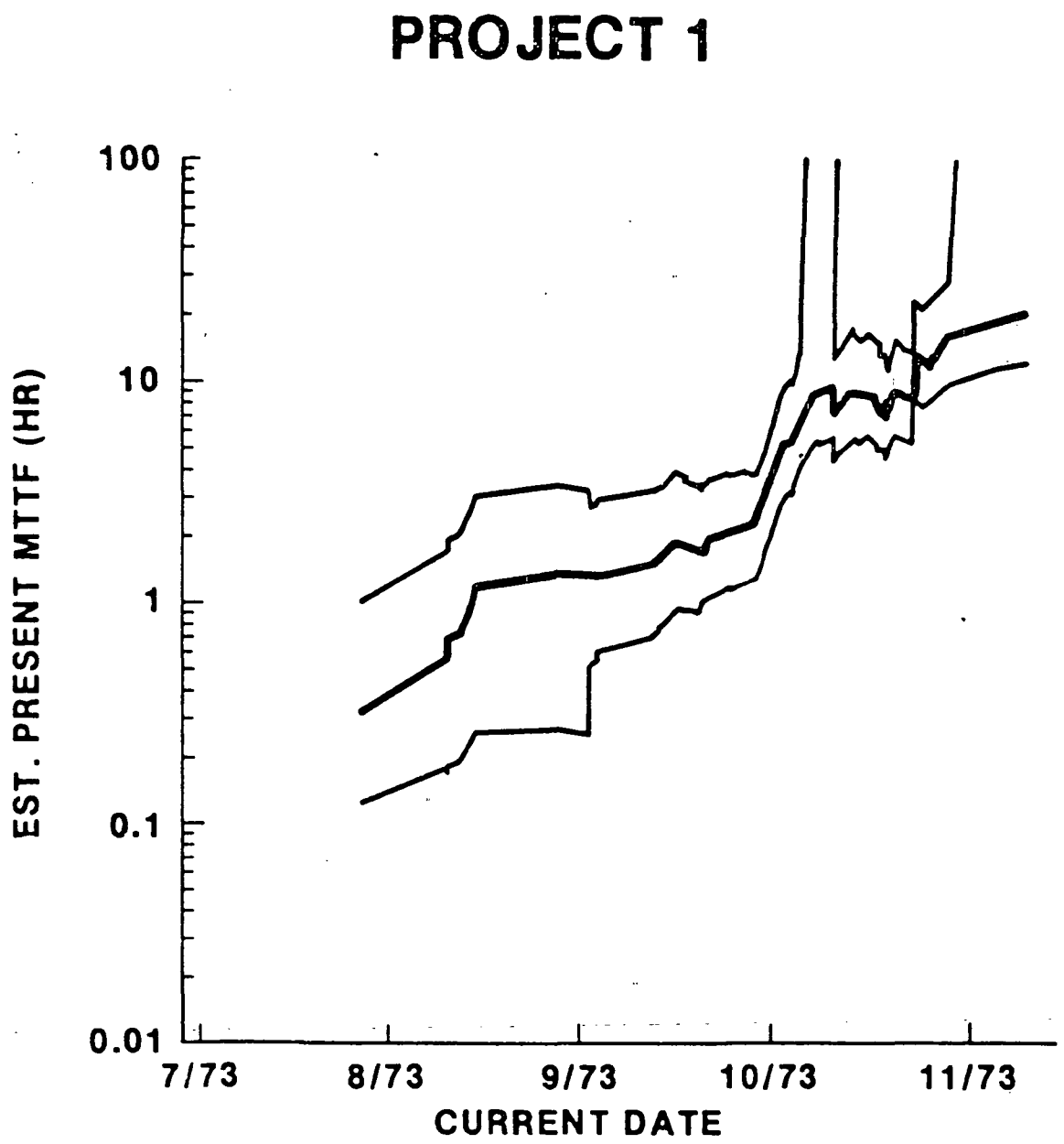
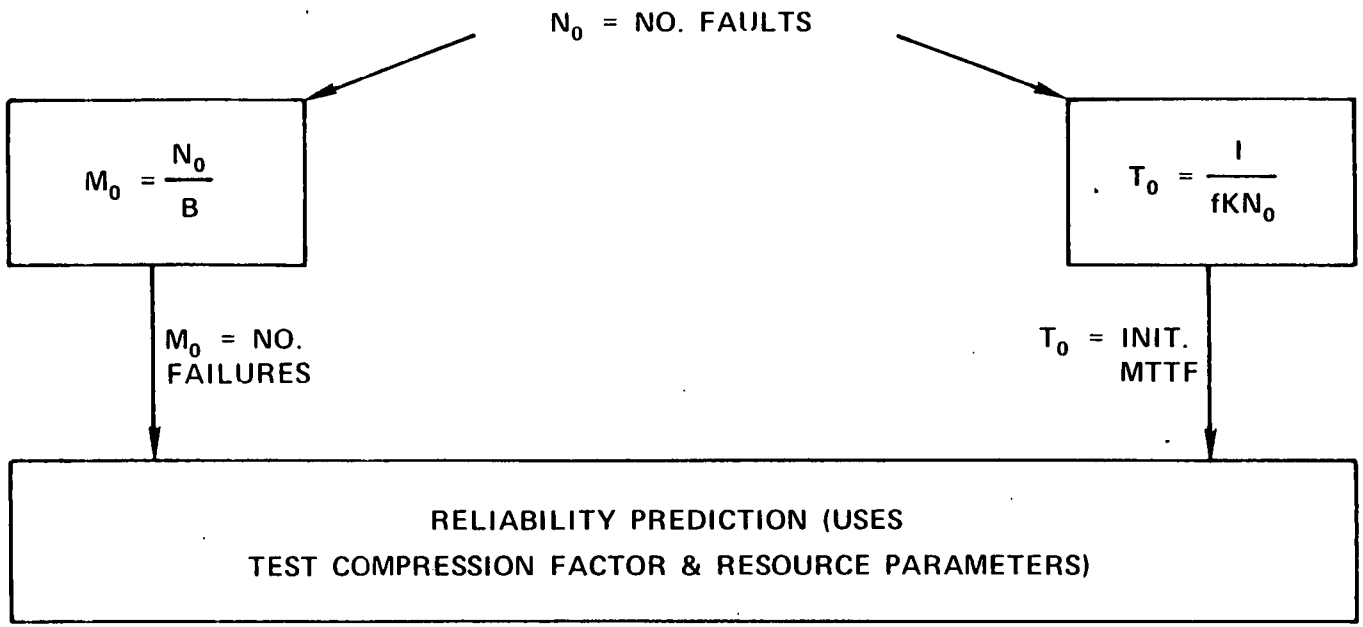


Figure 5. Software Reliability Estimation

SOFTWARE RELIABILITY PREDICTION



B = FAULT REDUCTION FACTOR

f = LINEAR EXEC. FREQ. OF PROGRAM

K = FAULT EXPOSURE RATIO

QUALITY OF INPUTS: BLACK→GOOD,GREEN→SATISFACTORY,RED→FAIR

J. Musa
Bell Lab.
11 of 11.

GENERAL ELECTRIC

General Electric Company
Command and Information Systems
450 Persian Drive
Sunnyvale, California

(408) 734-3571

A SIMULATION MODELING APPROACH TO UNDERSTANDING THE SOFTWARE DEVELOPMENT PROCESS

by

A. H. Stone
G. Y. Wong
J. A. McCall

Presented at the
Fourth Annual Software Engineering Workshop
November 19, 1979
Goddard Space Flight Center
Greenbelt, Maryland

A SIMULATION MODELING APPROACH TO UNDERSTANDING THE SOFTWARE DEVELOPMENT PROCESS

A. H. Stone
G. Y. Wong
J. A. McCall
Command and Information Systems
General Electric Company
Sunnyvale, California

ABSTRACT

This paper, resulting from research doen for the Air Force Office of Scientific Research (AFOSR), describes an assessment of the feasibility of utilizing simulation techniques to aid in the management of large-scale software developments. A model of the software development process was constructed, state-of-the-art prototype simulation tools used, and an experiment conducted to demonstrate the feasibility. A result of this effort is the concept of a Software Development Process Simulator which could be utilized to assist in project planning (cost estimation) and project control (progress status assessment).

INTRODUCTION

Significant progress has been made during the last few years in identifying the problems and complexities involved with the development of software systems and providing techniques to overcome these obstacles. What has evolved is a more disciplined environment for the production of software. Formal specification, design, and implementation methodologies are being developed. More milestones and visable software products during the development phases have been identified. Software support tools have become more sophisticated in providing assistance in the design and development of software. Considerable error and cost data have been collected and a better understanding of the software development environment is evolving. Cost, productivity, and reliability studies add to this understanding and provide data for prediction and estimation. The factors in software quality and associated metrics are being studied to obtain more quantitative measurements of the quality of a software product. Demonstration projects are being undertaken to prove the effectiveness of new techniques.

All of these R&D efforts contribute to a more disciplined and structured development process. This discipline and structure lends itself to more effecitve management. Most of the tools and techniques that have resulted from these R&D efforts support micro-level activities within the software development process. Few assist in the management of the entire process.

A potential management tool, made possible by the more disciplined approaches taken to software development, is a simulation model of the development process. Simulation models traditionally have been used by management for analyses such as system design studies, trade-off analyses, performance assessments, and impact analyses. A model of the software development process would facilitate these same types of analyses of the development effort itself. The analyses supported by such a tool would span both management planning (cost estimation) and control (progress and impact assessment).

The initial step toward developing a simulation tool to aid in the management of a software development involves developing the concept of such a tool and assessing the feasibility of using simulation techniques to construct a model of the software development process. This paper describes the results of this initial step. Specifically, under a contract sponsored by the Air Force Office of Scientific Research, the objectives were to:

- Determine the feasibility of applying simulation techniques to modeling the software development process.
- Describe the software development process in a manner conducive to developing a simulation model.
- Provide insights into modeling specific aspects of the software development process.
- Discuss the potential benefits and use of such a model.

MODELING APPROACH

A model is a representation of a system which gathers together in one place our understanding of the behavior of that system. The purpose of developing a model of a system is to have a vehicle for predicting the behavior of the system under various conditions. The adequacy of the model is normally determined by five criteria: (1) applicability – does the model answer the questions that we want to ask?; (2) confidence – is the model sufficiently accurate for our purposes?; (3) completeness – is the model broad enough to encompass all phenomena of interest?; (4) minimality – have system states that are unnecessarily discriminated been combined?; and, (5) independence – have system states that involve interacting factors been decomposed into multiple states?

The software development process has been modeled by researchers in software engineering primarily for the purpose of predicting the life cycle costs associated with developing computer software. The models that have been developed are macroscopic models which use analytic techniques to represent the behavior of a software development. However, where the analytic modeling approach treats the software development process as a "black box" process, the simulation modeling approach attempts to decompose the process and understand its internal behavior. With the simulation modeling approach, we view a software development organization as a collection of interdependent elements which act together in a collective effort to achieve the goal of implementing computer software. These elements are primarily personnel resources, such as programmers and analysts, and computer resources, such as terminals, computers and software tools.

Simulation modeling is the process of developing an internal representation and a set of transformation rules which can be used to predict the behavior of, and relationships between, the set of elements composing the system under study. The internal representation of a software development system is described by system state variables, such as software size and complexity, personnel productivity, and project status and progress. The transformation rules describe the interdependence between these system state variables. These transformation rules may be analytic – expressed in the form of functional relationships, or they may be representational – expressed in the form of an algorithm. Thus, the simulation approach provides for a microscopic view of the software development process.

The adequacy of the simulation approach of modeling the software development process is summarized in the following table.

Criteria	Evaluation
applicability	excellent
confidence	promising
completeness	excellent
minimality	excellent
independence	excellent

Applicability is excellent becuase a simulation model can be oriented toward studying any aspect of the software development process. Confidence is promising because if the accuracy of part of the model is not sufficient, then that part of the model can be expanded to a greater level of detail. Completeness is excellent because of the microscopic view that is taken with the simulation approach. Minimality and independence are both excellent because the feasibility inherent in the simulation approach allows system states to be combined or decomposed at the discretion of the modeler.

SOFTWARE DEVELOPMENT PROCESS MODEL

The challenges of managing a software development are immense because it is a multi-element process which is highly coupled and highly complex, and there are wide variations in controllable and uncontrollable variables between projects. In the past, the technique used by managers has been to decompose the software development process into "independent" subprocesses and manage those separately. This technique, generally following the Wolverton description [WOLV 74], does not usually reflect a very accurate model of the way software is currently being developed or is not in enough detail to analyze the causes of poor performance [TURN 76]. There has been recognition in recent years that interaction occurs, and that a continuous configuration management effort is required to keep the product of each phase up to date and consistent. Thus the traditional widely used "model" of the software development process is outmoded, no longer representing a true picture of how software is developed.

An accurate model of the software development process must account for several things. First, the idea that redesign, revisions to requirements, and changes to the source code take place constantly during the process, as the knowledge of the system evolves, must be acknowledged. Secondly, these revisions and corrections take place as a function of the activities the development personnel perform, not as a complete recycle of a phase; as evolution not revolution. Figure 1 is a representation of this evolution of knowledge on a timeline.

DECOMPOSITION OF THE SOFTWARE DEVELOPMENT PROCESS

Conceptually, the software development process is a process which is driven by a concept of, or requirement for, a target system and utilizes the resources of a software production factory to produce an operational system. The target system is a software system which has certain desired

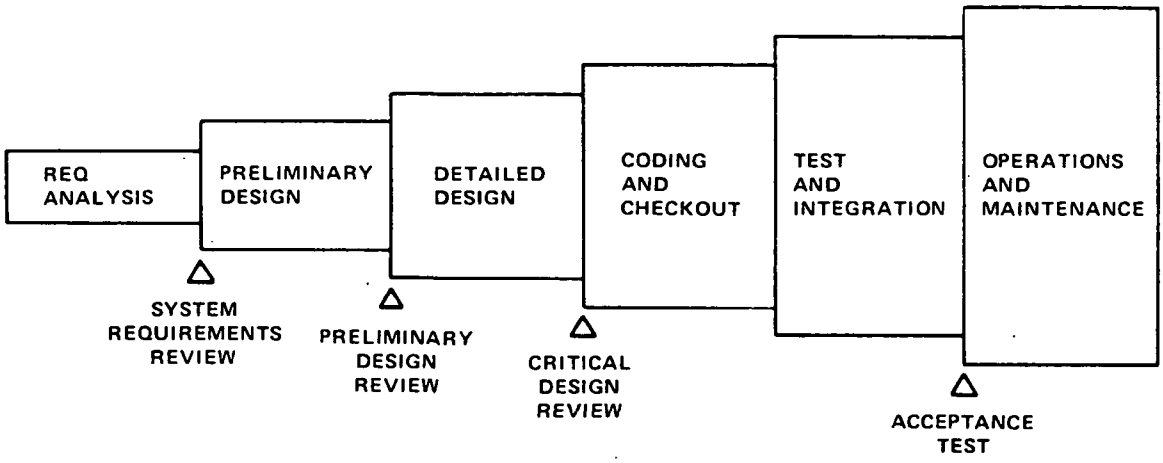


Figure 1. Timeline Representation of the Software Development Process

characteristics. These characteristics have an impact on the amount of resources which are consumed or utilized in the process of producing the operational system. The software production factory is the organizational, staffing, and development strategies superimposed on the resources of a project group (consisting of personnel and development tools), which provide the production capability and environment for accomplishing the system development. The operational system, the output of the process, is represented by the documents, data, and code produced as a result of the software development.

Imposed on this development process are a series of milestones which represent intermediate formal reviews of the progress towards the operational system. Further, there are documentation requirements which define what products are to be delivered. Almost all software developments have these milestone and documentation requirements. Perhaps the most rigorous set of requirements are those imposed by military standards.

Our approach to modelling the software development process was to decompose each of the phases in the high-level model described in the previous section into greater detail. The methodology used to accomplish this used military standards as a perspective and involved a three-dimensional view:

(1) Identification of the products of the software development process;

(2) Identification of the activities that comprise the process;

(3) Identification of the factors and resources that represent the target system and the software production factory.

Thus, we arrived at the model shown in Figure 2.

MODEL UTILITY

The conceptualization and decomposition of the software development process as a sequence of activities, as shown in Figure 2, provides a model which can be used at several levels. At one

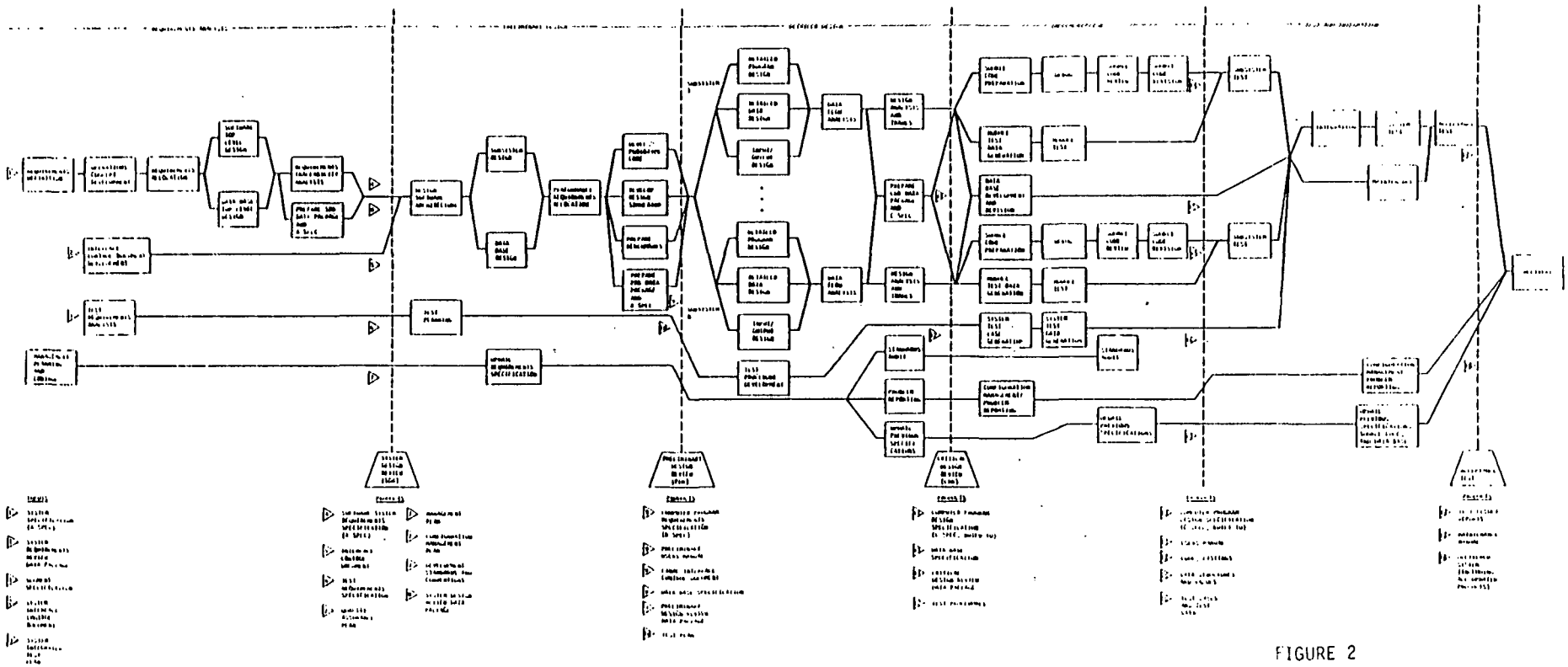


FIGURE 2

SOFTWARE DEVELOPMENT PROCESS MODEL

A. Stone
G.E.
6 of 24

level, the model can be used as a checklist for planning and progress status. The list of activities typically performed during a software development, and the interaction can be used to plan the activities to be performed in a future development. Once this plan is established, completion of these activities can be used as a status measurement more accurate than the normally imposed milestones.

At the next level, the model can be used as a PERT-COST tool. The current prototype tool that has been developed has the capability with which activity delay times could be modelled as distributions representing worst case, most likely, and best case estimates of the schedules for those activites. The simulation would then result in the calculation of the expected time in which the network of activities would be completed. An enhancement to the PERT-COST approach available with our simulation approach is modelling resource usage as a function of time also.

A third level, that at which the prototype simulator was developed, is a high level process model. At this level, the activities are modelled at a relatively high level. Sensitivities in the development plan and in the assumptions made in the model development could be analyzed. At a high level, impacts of using different techniques and tools could also be analyzed.

The last and most detailed level, is a detailed process model. At this level all of the concepts introduced in prior sections would be utilized to model the activities. The analysis capabilities possible in the process model mentioned above would be of greater fidelity due to the finer detail of the activity models. At this level of capability, the full complement of support to the management planning and control of a software development project would be provided.

EXPERIMENTING WITH THE MODEL

To further evaluate the feasibility of simulating the software development process, a prototype simulator was constructed and demonstrated by modeling a past software development. This prototype was developed with the idea in mind that it could eventually be extended to provide a full software development process simulation capability. In essence, this prototype enabled us to experiment with the basic concepts of the software development process model and provided some experience during which lessons could be learned and refinements in our approaches could be accomplished.

DESCRIPTION OF THE EXPERIMENT

The experiment was oriented toward modeling a past large-scale software system development. The simulated results were then compared with the historical data that was maintained about the development effort. This approach to an experiment is more modest than a full validation of the simulation model in which the simulated results would be used to predict the actual results, and a comparison of predicted versus actual would provide a validation criterion. Our experiment was more a calibration of the model to assess if, in fact, a development effort could be modelled to some degree of accuracy. Calibration is utilized by analytic techniques also (RCA PRICE-S and Putnam's SLIM) to tune the analytic model to the development organization. We envision this practice also pertaining to the Software Development Process Simulator, where various parameters or internal tables within the simulator could be tuned to a particular development organization by modeling past developments.

The software system development that was modelled consisted of three major subsystems (or CPCIs). The system was a command and control ground system developed under contract for the Air Force. The three subsystems ranged from 75,000 to 150,000 lines of JOVIAL source code each (including comments). Complete statistics on the development activity were maintained, including the number of design problem reports, software problem reports, and source code statistical profiles, as well as manpower expenditures.

RESULTS OF THE EXPERIMENT

Again, we were trying to calibrate our model and therefore were interested in achieving the highest possible agreement with the recorded development data. For this experiment, we examined actual manpower data from the simulation. The line labelled "ACTUAL" in Figure 3 is the graph of the data from the Air Force project superimposed on the graphs of the simulation results. Table 1 is a legend to be used with Figure 3. Cursory examination of the data in Figure 3 shows a clear correspondence between observed and experimental values.

Comparison of the graphs in Figure 3 shows that there is a 4.98% error between the areas, underneath the observed and experimental data curves. These results are considered quite acceptable. Some of the peaks of spikes seen in the actual data can be attributed to five-week fiscal months, which plotted at a granularity of one month causes higher manpower expenditures to be illustrated.

Resource	Title
SYSE	System Engineer
ANLT	Analyst
PROG	Programmer
QA	Quality Assurance Personnel

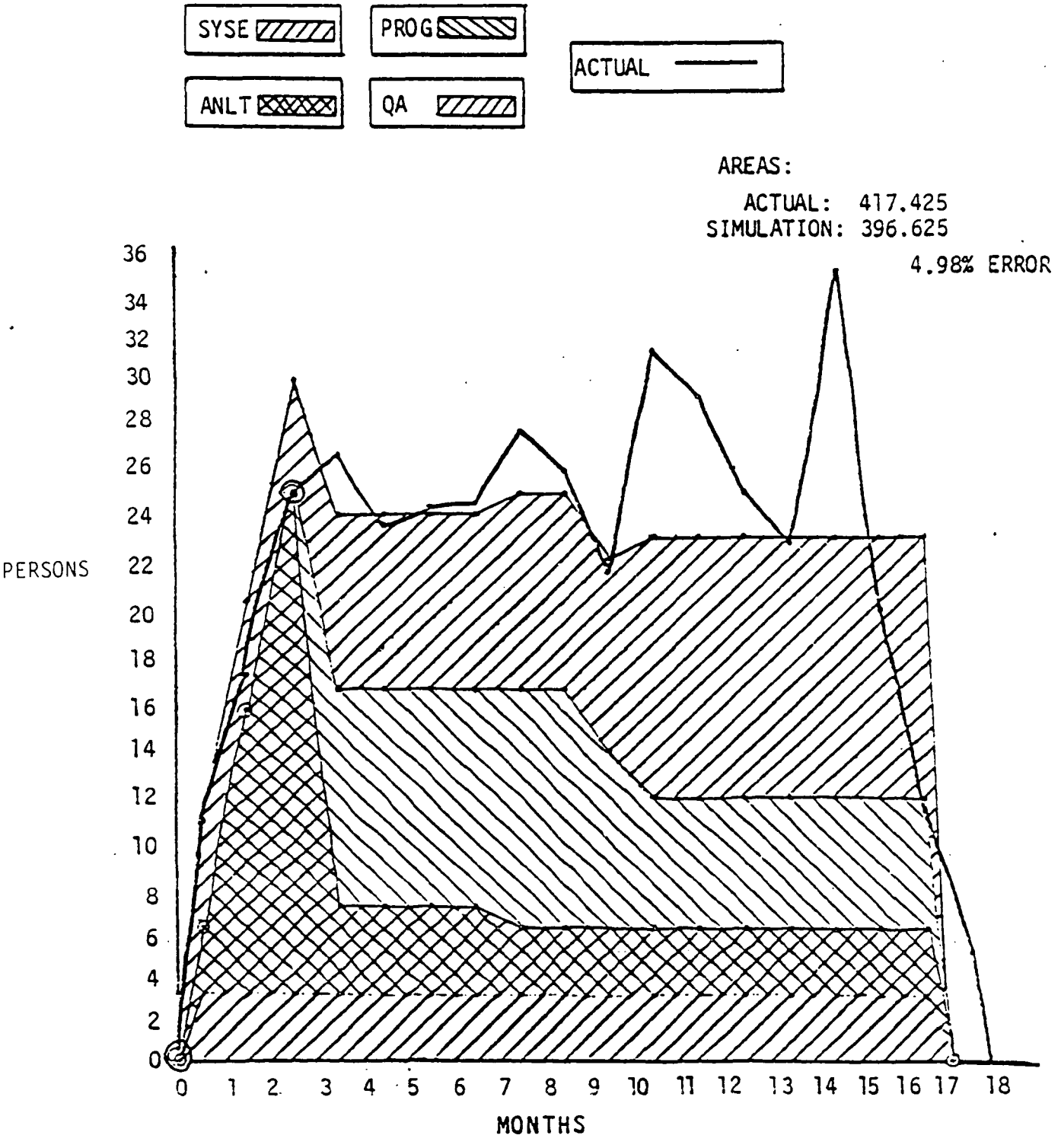


Figure 3 Experiment Results

CONCLUSIONS

Our research has covered the spectrum from concept formation to analysis to experimentation. First, we addressed the problem of how to apply simulation techniques to study the software development process. Then, we investigated what the characteristics of software developments are, based on the "world-view" established by our simulation approach. Finally, we studied when our modelling methodology is valid by learning how to design simulation experiments based on our modelling approach. Table 2 summarizes our major accomplishments, and indicates where we go from here.

Table 2

Conclusions Matrix

	Accomplished	Future
(1)	Simulation Approach	
	Combined activity-product model forms conceptual basis.	Identification of simulation variables, model rules, model inputs and outputs.
(2)	Process Decomposition	
	Activity-product network demonstrates practical application	Detailed specification of activities, products, factors, and resources.
(3)	Simulator Development	
	Simulator prototype demonstrates experimental feasibility.	Data collection to support full experiment.

REFERENCES

MCCA 79 McCall, J. A., et al., "A Simulation Modeling Approach to Understanding the Software Development Process," GE TIS 79CIS009, June 1979.

TURN 76 Turn, R., M. Davis, and R. Reinstedt, "A Management Approach to the Development of Computer-Based Systems," International Conference on Software Engineering, October 1976.

WOLV 74 Wolverton, Ray W., "The Cost of Developing Large-Scale Software," IEEE Transaction on Computers, Vol. 23, No. 6, 1974.

MAILING ADDRESS

Albert H. Stone
General Electric Company
Command and Information Systems
450 Persian Drive
Sunnyvale, California 94086
(408) 734-3571, x44

A SIMULATION MODELING APPROACH TO UNDERSTANDING THE SOFTWARE DEVELOPMENT PROCESS

A.H. STONE

GENERAL ELECTRIC COMPANY

COMMAND AND INFORMATION SYSTEMS

SOFTWARE TECHNOLOGIES GROUP

SUNNYVALE, CALIFORNIA

STUDY OBJECTIVES

AFOSR CONTRACT NO. F49620-78-C-0054

CONTRACT MONITOR: LT. COL. GEORGE McKEMIE

OBJECTIVES

- DETERMINE THE FEASIBILITY OF APPLYING SIMULATION TECHNIQUES TO MODELING THE SOFTWARE DEVELOPMENT PROCESS
- DESCRIBE THE SOFTWARE DEVELOPMENT PROCESS IN A MANNER CONDUCIVE TO DEVELOPING A SIMULATION MODEL
- PROVIDE INSIGHTS INTO MODELING SPECIFIC ASPECTS OF THE SOFTWARE DEVELOPMENT PROCESS
- DISCUSS THE POTENTIAL BENEFITS AND USE OF SUCH A MODEL

WHY SIMULATION?

COMPARING TECHNIQUES:

CRITERIA	TECHNIQUE EVALUATION	
	ANALYTIC	SIMULATION
APPLICABILITY	LIMITED	EXCELLENT
CONFIDENCE	MARGINAL	PROMISING
COMPLETENESS	LIMITED	EXCELLENT
MINIMALITY	EXELLENT	EXCELLENT
INDEPENDENCE	LIMITED	EXCELLENT

IN ADDITION, SIMULATION:

- HAS DEMONSTRATED SUCCESS AT MODELING PEOPLE PROCESSES AND MAN-MACHINE INTERACTIONS
- ALLOWS FLEXIBLE YET DETAILED MODELING
- PROVIDES A USABLE TESTBED FOR EVALUATING THE MODEL
- PROVIDES A VEHICLE FOR MEANINGFUL "WHAT IF" ANALYSES

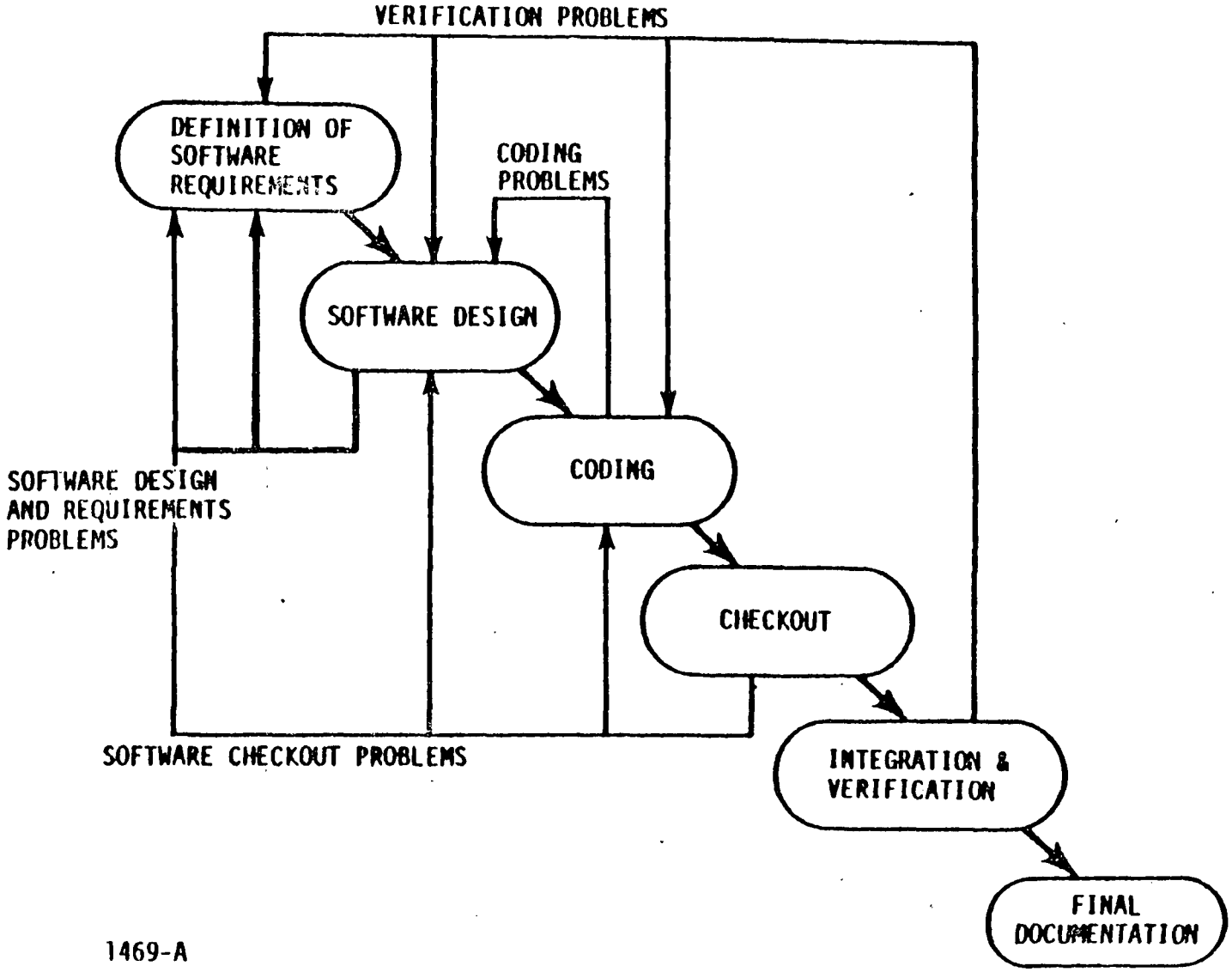
MOTIVATION FOR UNDERSTANDING

TOOLS ARE NEEDED FOR

- PROJECT PLANNING
  - COST ESTIMATION
  - TIME REQUIREMENTS
  - RESOURCE REQUIREMENTS
- PROJECT CONTROL
  - PERFORMANCE ASSESSMENT
  - BOTTLENECK ANALYSIS
  - RESOURCE TRADEOFF ANALYSIS
- TECHNOLOGY ASSESSMENT
  - ASSESSMENT OF IMPACT OF NEW TOOLS, TECHNOLOGIES, AND METHODOLOGIES
- CONTINGENCY PLANNING
  - IMPACT ASSESSMENT

A. Stone
G.E.
14 of 24

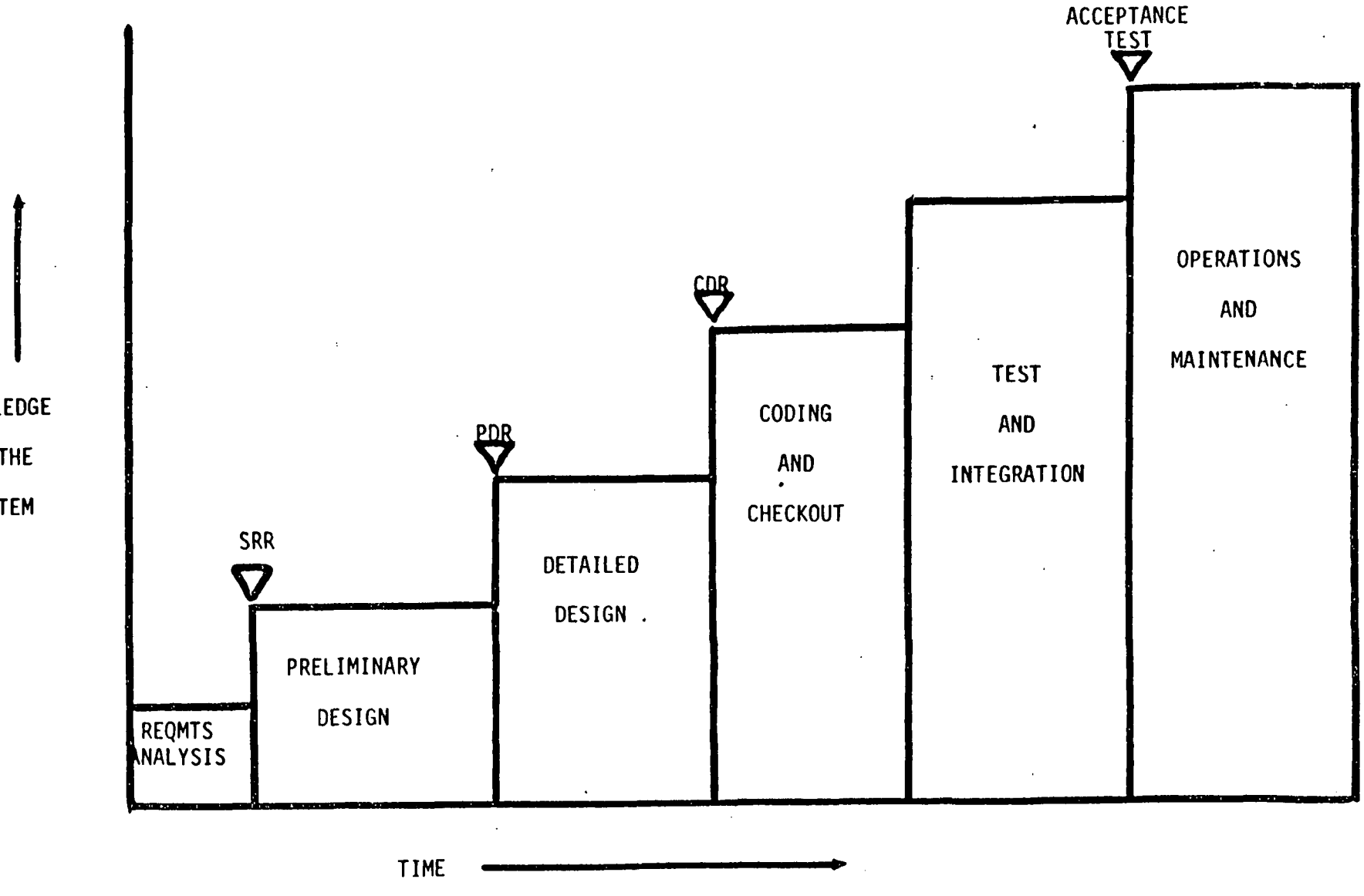
SOFTWARE DEVELOPMENT PROCESS



1469-A

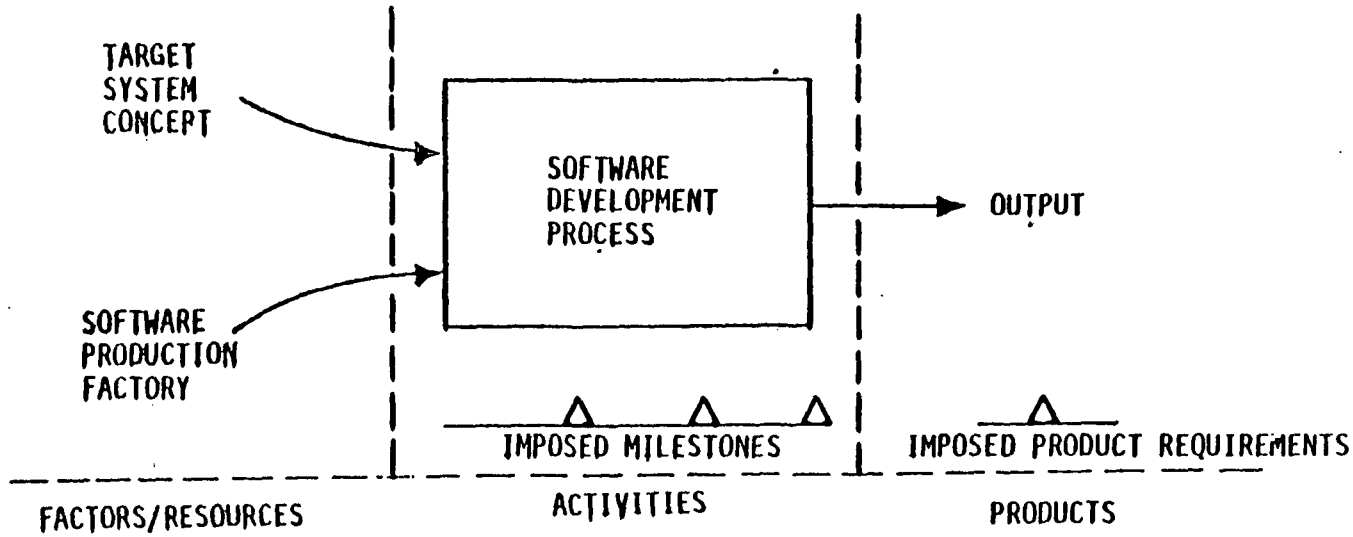
A. Stone
G.E.
15 of 24

SOFTWARE DEVELOPMENT PROCESS MODEL



A. Stone
G.E.
16 of 24

SOFTWARE DEVELOPMENT PROCESS CONCEPT

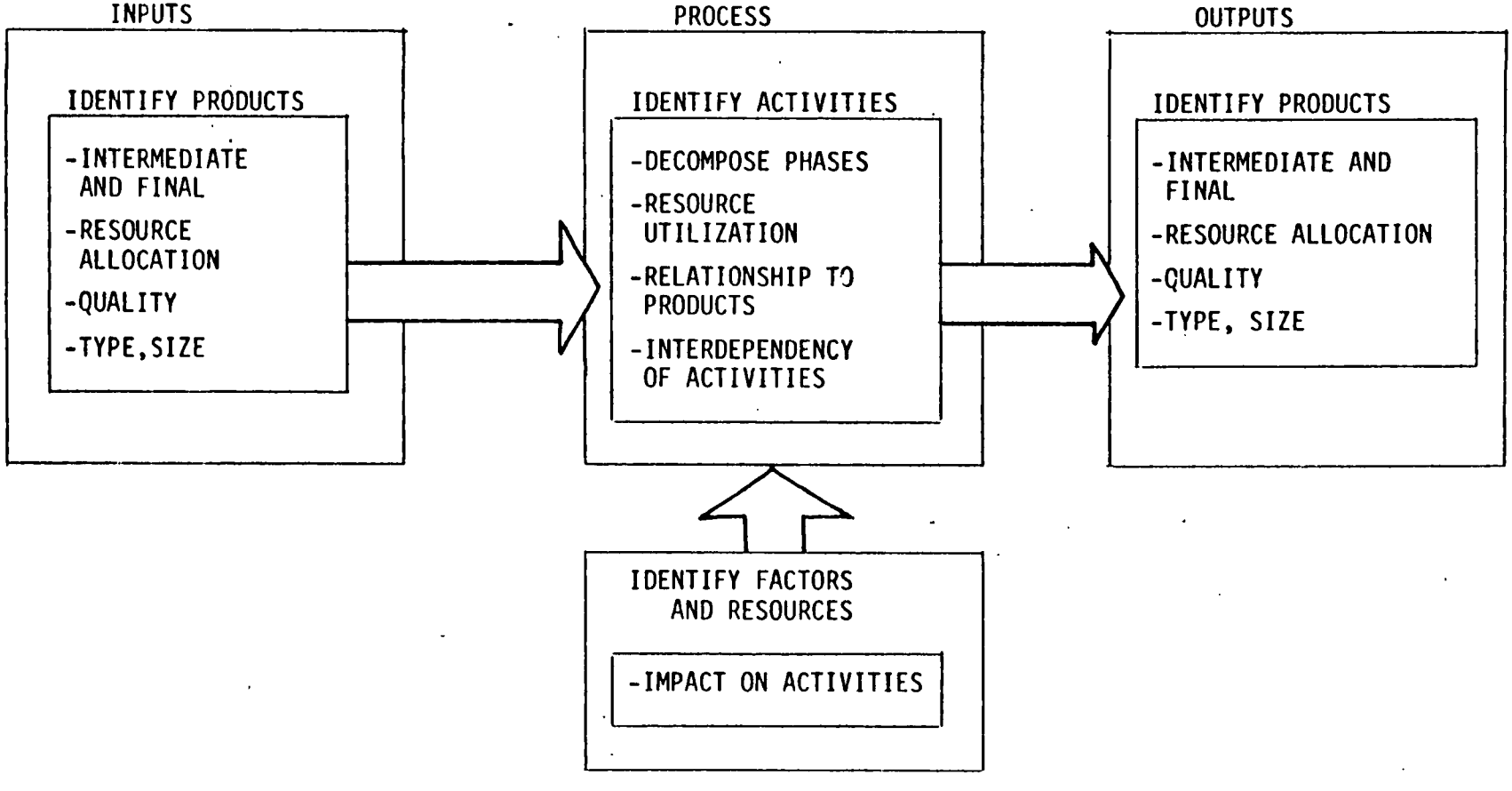


1781A

A. Stone
G.E.
17 of 24

DECOMPOSITION OF THE SOFTWARE DEVELOPMENT PROCESS

DECOMPOSITION METHODOLOGY



THE BASIC CONCEPTS OF SDPS

SYSTEM DEVELOPMENT PROGRESS IS REPRESENTED BY:

- THE MIX OF ONGOING DEVELOPMENT ACTIVITIES (ACTIVITY OR WORK BREAKDOWN STRUCTURE MODEL)
- THE EVOLUTION OF SYSTEM KNOWLEDGE IN THE FORM OF PRODUCTS (PRODUCT PROGRESSION MODEL)

HOW MUCH PROGRESS IS SHOWN BY:

- PERSON HOURS BY TYPE OF PERSON FOR EACH ACTIVITY
- LINES OF DOCUMENTATION OR CODE FOR PRODUCTS

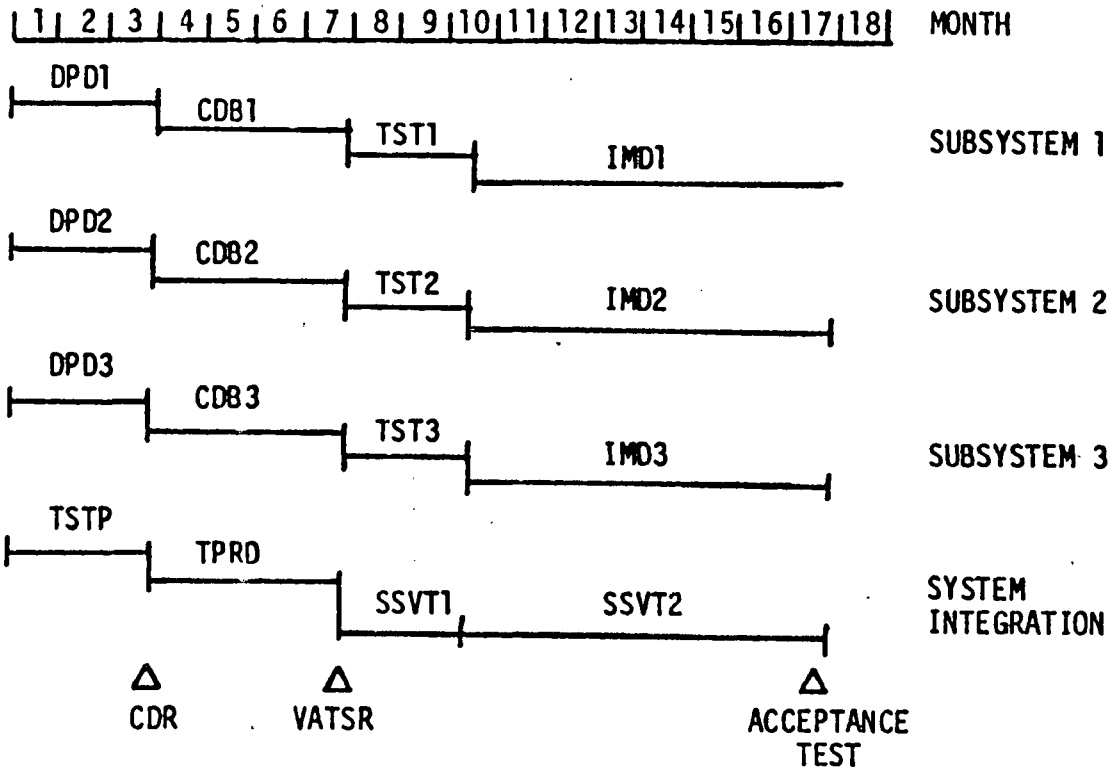
QUALITY OF THE EFFORT IS REFLECTED BY:

- PERSONNEL EXPERIENCE IN AN ACTIVITY
- QUALITY METRICS VALUES FOR EACH PRODUCT

MODEL UTILITY

- CHECKLIST
- PERT-COST
- PROCESS MODEL
- DETAILED PROCESS MODEL

EXPERIMENT DESCRIPTION



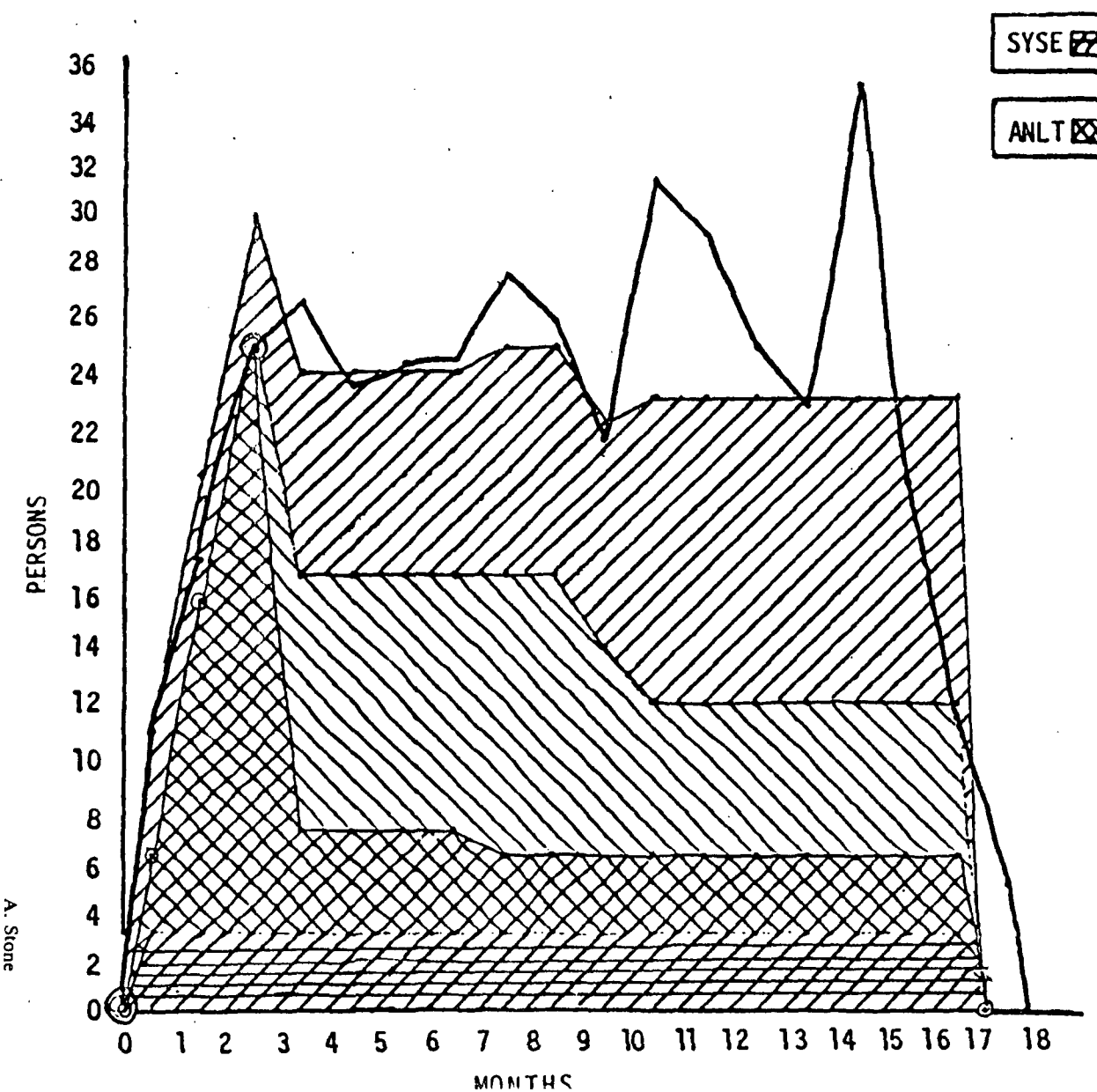
	LEGEND
DPDi	DETAILED PROGRAM DESIGN
CDBi	CODING AND DEBUG
TSTi	SUBSYSTEM TEST
IMDi	INTEGRATION & MAINTENANCE SUPPORT
TSTP	TEST PLAN PREPARATION
TPRD	TEST PRUCEDURE DEVELOPMENT
SSVT1	SYSTEM TEST & INTEGRATION
SSVT1	
CDR	CRITICAL DESIGN REVIEW
VATSR	VALIDATION & ACCEPTANCE TEST SPECIFICATION REVIEW

PROCESS FLOW:

```
05/29/79                    19.6930                                PAGE     1

                PATH EXPRESSION PARSER          VERSION 1.5

 1      BEGIN SAMPLE
 2      ;
 3      ;    SDPS SAMPLE MODEL
 4      ;
 5      ;         BY   J.A. MCCALL
 6      ;              A.H. STONE
 7      ;
 8      ;         APRIL 1979
 9      ;
10           MACRO-TABLE
11              DESIGN = (DPD1:DPD2:DPD3)
12              CODING = (CDB1:CDB2:CDB3)
13              TEST   = (TST1:TST2:TST3)
14              INTEG  = (IMD1:IMD2:IMD3)
15           END-MACRO
16      ;
17      ;    BEGIN THE SIMULATION
18      ;
19           SIMULATE = (DESIGN:TSTP,          ; DESIGN PHASE
20                       CODING:TPRD,          ; CODING PHASE
21                       TEST:SSVT1,           ; TESTING PHASE
22                       INTEG:SSVT2)          ; INTEGRATION PHASE
23      ;
24      END  ; SAMPLE

              24 LINES PROCESSED
               0 NON-FATAL ERRORS
               0 FATAL ERRORS

--- PROCESSING COMPLETED
```

ISP

EXPERIMENT RESULTS



AREAS:

ACTUAL: 417.425
SIMULATION: 396.625

4.98% ERROR

A. Stone
G.E.
23 of 24

CONCLUSIONS/FURTHER INVESTIGATIONS

CONCLUSIONS MATRIX

	ACCOMPLISHED	FUTURE
(1) SIMULATION APPROACH	COMBINED ACTIVITY-PRODUCT MODEL FORMS CONCEPTUAL BASIS.	IDENTIFICATION OF SIMULATION VARIABLES, MODEL RULES, MODEL INPUTS AND OUTPUTS.
(2) PROCESS DECOMPOSITION	ACTIVITY-PRODUCT NETWORK DEMONSTRATES PRACTICAL APPLICAITON.	DETAILED SPECIFICATION OF ACTIVITIES, PRODUCTS, FACTORS, AND RESOURCES.
(3) SIMULATOR DEVELOPMENT	SIMULATOR PROTOTYPE DEMONSTRATES EXPERIMENTAL FEASIBILITY.	DATA COLLECTION TO SUPPORT FULL EXPERIMENT