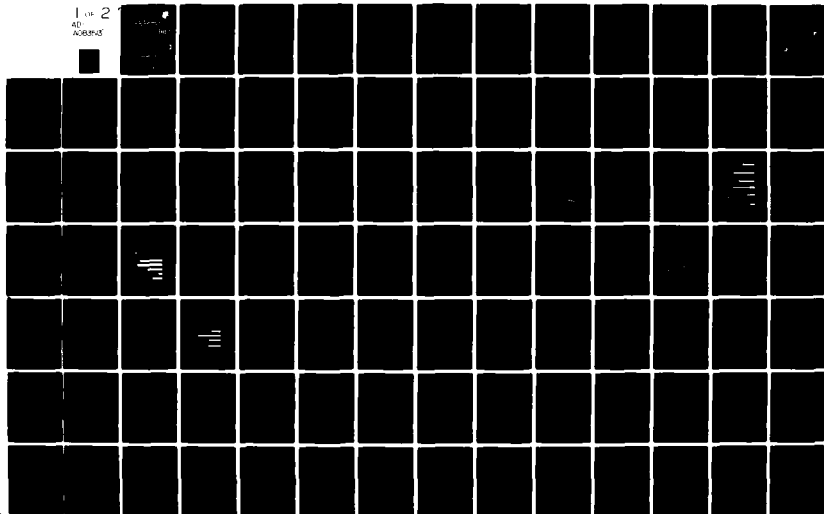


AD-A083 513

GENERAL ELECTRIC CO ARLINGTON VA F/G 9/2  
A MATCHED PROJECT EVALUATION OF MODERN PROGRAMMING PRACTICES. V--ETC(U)  
FEB 80 P WILLIMAN, B CURTIS F30602-77-C-0194  
UNCLASSIFIED 791SP006 RADC-TR-80-6-VOL-2 NL

1 of 2  
AD-  
SUBMIT



RADC-TR-80-6, Vol II (of two)  
Interim Report  
February 1980

12  
NW



ADA 083513

# A MATCHED PROJECT EVALUATION OF MODERN PROGRAMMING PRACTICES Scientific Report on the ASTROS Plan

General Electric Company

Phil Milliman  
Bill Curtis

# LEVEL III

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

DTIC  
ELECTE  
S APR 28 1980 D  
E

**ROME AIR DEVELOPMENT CENTER**  
**Air Force Systems Command**  
**Griffiss Air Force Base, New York 13441**

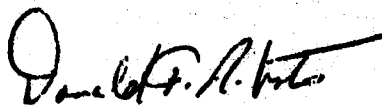
DDC FILE COPY

80 4 25 015

This report has been reviewed by the RADC Public Affairs Office (PA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

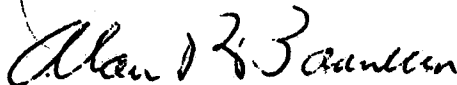
RADC-TR-80-6, Vol II (of two) has been reviewed and is approved for publication.

APPROVED:



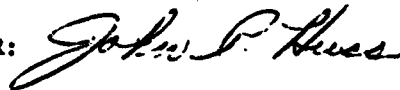
DONALD F. ROBERTS  
Project Engineer

APPROVED:



ALAN R. BARNUM, Asst Chief  
Information Sciences Division

FOR THE COMMANDER:



JOHN P. HUSS  
Acting Chief, Plans Office

If your address has changed or if you wish to be removed from the RADC mailing list, or if the addressee is no longer employed by your organization, please notify RADC (ISIS), Griffiss AFB NY 13441. This will assist us in maintaining a current mailing list.

Do not return this copy. Retain or destroy.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

19 REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM	
1. REPORT NUMBER RADC-TR-80-6 Vol-2 (of two)	2. GOVT ACCESSION NO. AD-A083 513	3. RECIPIENT'S CATALOG NUMBER	
4. TITLE (and Subtitle) A MATCHED PROJECT EVALUATION OF MODERN PROGRAMMING PRACTICES, Volume II, Scientific Report on the ASTROS Plan.		5. TYPE OF REPORT & PERIOD COVERED Interim Report, 1 Sep 77 - 30 Nov 78,	
6. AUTHOR(s) Phil Milliman Bill Curtis		7. PERFORMING ORG. REPORT NUMBER 791SP006	
9. PERFORMING ORGANIZATION NAME AND ADDRESS General Electric, Suite 200 1755 Jefferson Davis Highway Arlington VA 22202		8. CONTRACT OR GRANT NUMBER(s) F30602-77-C-0194	
11. CONTROLLING OFFICE NAME AND ADDRESS Rome Air Development Center (ISIS) Griffiss AFB NY 13441		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 63728F 25280103	
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office) Same		12. REPORT DATE February 1980	
15. SECURITY CLASS. (of this report) UNCLASSIFIED		13. NUMBER OF PAGES 123	
15a. DECLASSIFICATION DOWNGRADING SCHEDULE N/A		16. DISTRIBUTION STATEMENT (of this Report)  Approved for public release; distribution unlimited.	
17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)  Same		18. SUPPLEMENTARY NOTES RADC Project Engineer: Donald E. Roberts (ISIS)	
19. KEY WORDS (Continue on reverse side if necessary and identify by block number) Modern Programming Practices Chief Programmer Teams Software Science Software Error Data Software Management		Software Data Base Matched Project Evaluation Structured Programming	
20. ABSTRACT (Continue on reverse side if necessary and identify by block number) Rome Air Development Center (RADC) and the Space and Missile Test Center (SAM-TEC) at Vandenberg Air Force Base have jointly developed a plan for improving software development called Advanced Systematic Techniques for Reliable Operational Software (ASTROS). This system provides guidelines for applying the following modern programming practices to software development: <i>→ next page</i>			

(Cont'd)

DD FORM 1 JAN 73 1473

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

4074/6

- Structured design and testing,
- HIPO charts,
- Chief programmer teams,
- Structured coding,
- Structured walk-throughs,
- Program support library.

In order to test the utility of these techniques, two development projects (non-real-time), sponsored by SAMTEC from the Metric Integrated Processing System, were chosen for a quasi-experimental comparison. This system provides control and data analysis for missile launches. The Launch Support Data Base (LSDB) was developed under the guidelines of the ASTROS plan, while the Data Analysis Processor (DAP) was developed using conventional techniques.

This report is the first of two volumes describing the performance of the LSDB project implemented under ASTROS. This volume presents a condensed report of the results of this study which has been prepared for a non-scientific and/or managerial audience. Readers wanting a detailed account of the theories and analyses underlying these results, are referred to Volume II. Where material has been substantially condensed in this volume, a reference will be provided to the corresponding section in Volume II where further elaboration of the material is available.

The performance of the LSDB project was comparable to that of similar sized software development projects on numerous criteria. The amount of code produced per man-month was typical of conventional development efforts. Nevertheless, the performance of the LSDB project was superior to that of the DAP project. Thus, the benefits of the modern programming practices employed on the LSDB project were limited by the constraints of environmental factors such as computer access and turnaround time.

While the results of this study demonstrated reduced programming effort and improved software quality for a project guided by modern programming practices, no causal interpretation can be reliably made. That is, with only two projects and no experimental controls, causal factors can only be suggested, not isolated. The generalizability of these results to other projects is uncertain. The data also do not allow analyses of the relative values of each separate practice. Nevertheless, the results of this study suggest that future evaluations will yield positive results if constraints in the development environment are properly controlled.

Accession For	
NA	<input checked="" type="checkbox"/>
DA	<input type="checkbox"/>
CS	<input type="checkbox"/>
...	<input type="checkbox"/>
...	<input type="checkbox"/>
...	<input type="checkbox"/>
...	<input type="checkbox"/>

Dist	For
A	al

## TABLE OF CONTENTS

<u>Title</u>	<u>Page</u>
Abstract . . . . .	ii
Table of Contents . . . . .	iv
List of tables . . . . .	vi
List of figures . . . . .	vii
1. Introduction . . . . .	1
1.1 Purpose for this research . . . . .	1
1.2 Advanced Systematic Techniques for Reliable Operational Software (ASTROS) . . . . .	2
1.2.1 Structured design and testing . . . . .	2
1.2.2 HIPO charts . . . . .	3
1.2.3 Chief programmer teams . . . . .	3
1.2.4 Structured coding . . . . .	5
1.2.5 Structured walk-throughs . . . . .	5
1.2.6 Program support library . . . . .	7
1.3 Metric Integrated Processing System (MIPS) . . . . .	8
1.3.1 Launch Support Data Base (LSDB) . . . . .	8
1.3.2 Data Analysis Processor (DAP) . . . . .	9
1.4 Project Environment . . . . .	9
1.4.1 Hardware . . . . .	9
1.4.2 Software . . . . .	9
1.4.3 Training . . . . .	10
1.5 Data Acquisition . . . . .	10
2. Theories Relevant to Software Development . . . . .	15
2.1 Putnam's Software Life Cycle Model . . . . .	15
2.2 Halstead's Software Science . . . . .	16
2.2.1 Volume . . . . .	16
2.2.2 Level . . . . .	17
2.2.3 Effort . . . . .	17
2.2.4 Scope . . . . .	17
2.2.4.1 Minimum scope case . . . . .	18
2.2.4.2 Maximum scope case . . . . .	18
2.2.4.2.1 Lower maximum scope . . . . .	18
2.2.4.2.2 Upper maximum scope . . . . .	19
2.2.5 Delivered Errors . . . . .	19
2.3 McCabe's Complexity Metric . . . . .	20
2.4 Software Quality Metrics . . . . .	22
3. Data Analysis . . . . .	25
3.1 Comparative Analyses: LSDB Versus DAP . . . . .	26
3.1.1 Descriptive data . . . . .	26
3.1.2 Level of Technology: Putnam's Model . . . . .	29
3.1.3 Programming Scope and Time: Halstead's Model . . . . .	32
3.1.4 Complexity of Control Flow: McCabe's Model . . . . .	34
3.1.5 Software Quality Metrics . . . . .	37
3.1.6 Comparison to RADC Database . . . . .	40

<u>Title</u>	<u>Page</u>
3.2 Error Analyses . . . . .	40
3.2.1 Error Categories . . . . .	40
3.2.1.1 Descriptive data . . . . .	40
3.2.1.2 Comparison with TRW data . . . . .	42
3.2.2 Prediction of Errors at the Subsystem Level . . . . .	42
3.2.3 Error Trends over Time . . . . .	46
3.2.4 Post-Development Errors . . . . .	49
4. Interviews . . . . .	52
4.1 Chief Programmer Teams . . . . .	52
4.2 Design and Coding Practices . . . . .	55
5. Conclusions . . . . .	58
5.1 Current Results . . . . .	58
5.2 Future Approaches to Evaluative Research . . . . .	60
References . . . . .	61
Appendices	
A. Data Collection Forms . . . . .	65
B. RADC Data Base Scatterplots . . . . .	89
C. Proof of Relationships Among Levels of Programming Scope . . . . .	103
D. Software Quality Metrics . . . . .	107
E. Manhours by Month and Work Breakdown Structures for LSDB and DAP . . . . .	113
F. Number of Errors by Category and Month for LSDB . . . . .	117

LIST OF TABLES

<u>Table</u>	<u>Title</u>	<u>Page</u>
1	Data Collection Forms . . . . .	12
2	Variables in the RADC Database . . . . .	14
3	Software Quality Factors . . . . .	24
4	Number of lines by Subsystem: LSDB versus DAP . . . . .	27
5	Volume, Level, and Effort by Levels of Scope for Subsystems from Each Project . . . . .	33
6	McCabe Values for Selected Subroutines from Each Project . . . . .	36
7	Values of Software Quality Metrics for Selected Subroutines . . . . .	39
8	Frequencies of Error Categories . . . . .	43
9	Frequencies of Runs and Errors by Subsystem . . . . .	45



## LIST OF FIGURES

<u>Figures</u>	<u>Title</u>	<u>Page</u>
1	HIPO chart . . . . .	4
2	Control Structures allowed in structured programming . .	6
3	Control flow graphs and associated values for $v(G)$ . . .	21
4	Decomposition of a control flowgraph to essential complexity . . . . .	23
5	Chronological manpower loading for LSDB and DAP . . . .	28
6	Chronological history of LSDB and DAP by phase . . . . .	30
7	Percentage of effort by development phase . . . . .	31
8	Predictions from Halstead's model of time to code both projects . . . . .	35
9	An example from LSDB of decomposition to essential complexity . . . . .	38
10	Regression of delivered source lines of code on productivity . . . . .	41
11	Comparison of error distributions between LSDB and three TRW studies . . . . .	44
12	Action reports, error runs, and post-development test errors by month . . . . .	47
13	Error ratio by month . . . . .	48
14	Prediction of post-development errors. . . . .	50

## 1. INTRODUCTION

### 1.1 Purpose for This Research

In 1973 Boehm chronicled the tremendous impact of software on the cost and reliability of advanced information processing systems. Recently, DeRoze and Nyman (1978) estimated the yearly cost for software within the Department of Defense to be as large as three billion dollars. De Roze (1977) reported that perhaps 115 major defense systems depend on software for successful operation. Nevertheless, the production and maintenance of software for Defense is frequently inefficient.

In an effort to improve both software quality and the efficiency of software development and maintenance, a number of techniques have been developed as alternatives to conventional programming practices. These modern programming practices include such techniques as structured coding, structured design, program support libraries, and chief programmer teams (W. Myers, 1978; Tausworthe, 1979). The New York Times project implemented by IBM (Baker, 1972) was lauded as a successful initial demonstration of these techniques. Yet, some problems appear to have resulted from an early release of the system (Yourdon Report, 1976). Considerable variability has been reported in subsequent studies on the effect of these techniques for various project outcomes (Belford, Donahoo, & Heard, 1977; Black, 1977; Brown, 1977; Walston & Felix, 1977). Many evaluations have rested solely on subjective opinions obtained in questionnaires. There is a critical need for empirical research evaluating the effects of modern programming practices on software development projects.

Rome Air Development Center (RADC) and the Space and Missile Test Center (SAMTEC) at Vandenberg Air Force Base have jointly developed a plan for improving software development called ASTROS - Advanced Systematic Techniques for Reliable Operational Software (Lyons & Hall, 1976). This plan describes several modern programming practices which should result in more reliable and less expensively produced and maintained software. In order to evaluate the utility of these techniques, RADC is sponsoring research into their use on a software development project at SAMTEC.

Two development projects from the non-real-time segment of the Metric Integrated Processing System (MIPS) were chosen for comparison. The MIPS system provides control and data analysis in preparation for missile launches. The Launch Support Data Base (LSDB) segment of MIPS was implemented

under the ASTROS plan while the Data Analysis Processor (DAP) was implemented with conventional software development techniques.

Data from LSBDB were compared with results from DAP, allowing a quasi-experimental comparison of two similar projects in the same environment, one implemented under the ASTROS plan. Data from these two projects were also compared with the results from software projects elsewhere in industry. Because there is little control over many of the influences on the two projects, a causal relationship between programming practices and performance cannot be proved, but the data can be investigated for evidence of their effects.

## 1.2 Advanced Systematic Techniques for Reliable Operational Software (ASTROS)

The ASTROS plan was a joint effort by SAMTEC and RADC to implement and evaluate modern programming practices in an Air Force operational environment. ASTROS applied these practices to selected programming projects in order to demonstrate empirically the premise that these techniques would yield lower costs per line of code, more reliable software, more easily maintained code, and less schedule slippage.

The ASTROS project focused on three objectives: 1) an investigation and validation of structured programming tools and concepts, 2) improving management aspects of structured programming, and 3) empirical measurement of project process and outcomes. The core of the ASTROS plan was the specification of a set of modern programming practices. The implementation of these practices by the LSBDB project team as described by Salazar and Hall (1977) is discussed below.

1.2.1 Structured design and testing - is the practice of top-down development or stepwise refinement (Stevens, Myers, & Constantine, 1974; Yourdon & Constantine, 1979). Each subsystem is designed from the control sections down to the lowest level subroutines prior to the start of coding. Thus, the highest level units of a system or subsystem are coded and tested first. Top-down implementation does not imply that all system components at each level must be finished before the next level is begun, but rather the fathers of a unit must be completed before this unit can be coded.

Since higher level units will normally invoke lower level units, dummy code must be substituted temporarily for these latter units. The required dummy units (program stubs) may be generalized, prestored on disk, and included

automatically by the linkage editor during a test run, as in the case of a CALL sequence. Although program stubs normally perform no meaningful computations, they can output a message for debugging purposes each time they are executed. Thus, it is possible to exercise and check the processing paths in the highest level unit before initiating implementation of the lower level units it invokes. This procedure is repeated, substituting actual program units for the dummy units at successively lower levels until the entire system has been integrated and tested. Program units at each level are fully integrated and tested before coding begins at the next lower level.

1.2.2 HIPO charts - Hierarchical Input-Process-Output charts are diagrammatic representations of the operations performed on the data by each major unit of code (Katzen, 1976; Stay, 1974). A HIPO chart is essentially a block diagram showing the inputs into a functional unit, the processes performed on that data within the unit, and the output from the unit (Figure 1). There was one HIPO per functional unit, with the processing in one unit being expanded to new HIPOs until the lowest level of detail was reached. The hierarchical relationships among the HIPO charts are displayed in a Visual Table of Contents.

1.2.3 Chief programmer teams - are organized so that functional responsibilities such as data definition, program design, and clerical operations are assigned to different members (Baker, 1972; Baker & Mills, 1973; Barry & Naughton, 1975). This approach results in better integration of the team's work, avoiding the isolation of individual programmers that has often characterized programming projects. The chief programmer team is made up of 3 core members and optional support members who are programmers.

- Chief programmer - is responsible to the project manager for developing the system and managing the programming team. He carries technical responsibility for the project including production of the critical core of the programming system in detailed code, direct specification of all other codes required for system implementation, and review of the code integration.
- Back-up programmer - supports the chief programmer at a detailed task level so that he can assume the chief programmer's role temporarily or permanently if required. In the LSDB project, he was responsible for generating 80% of the code.

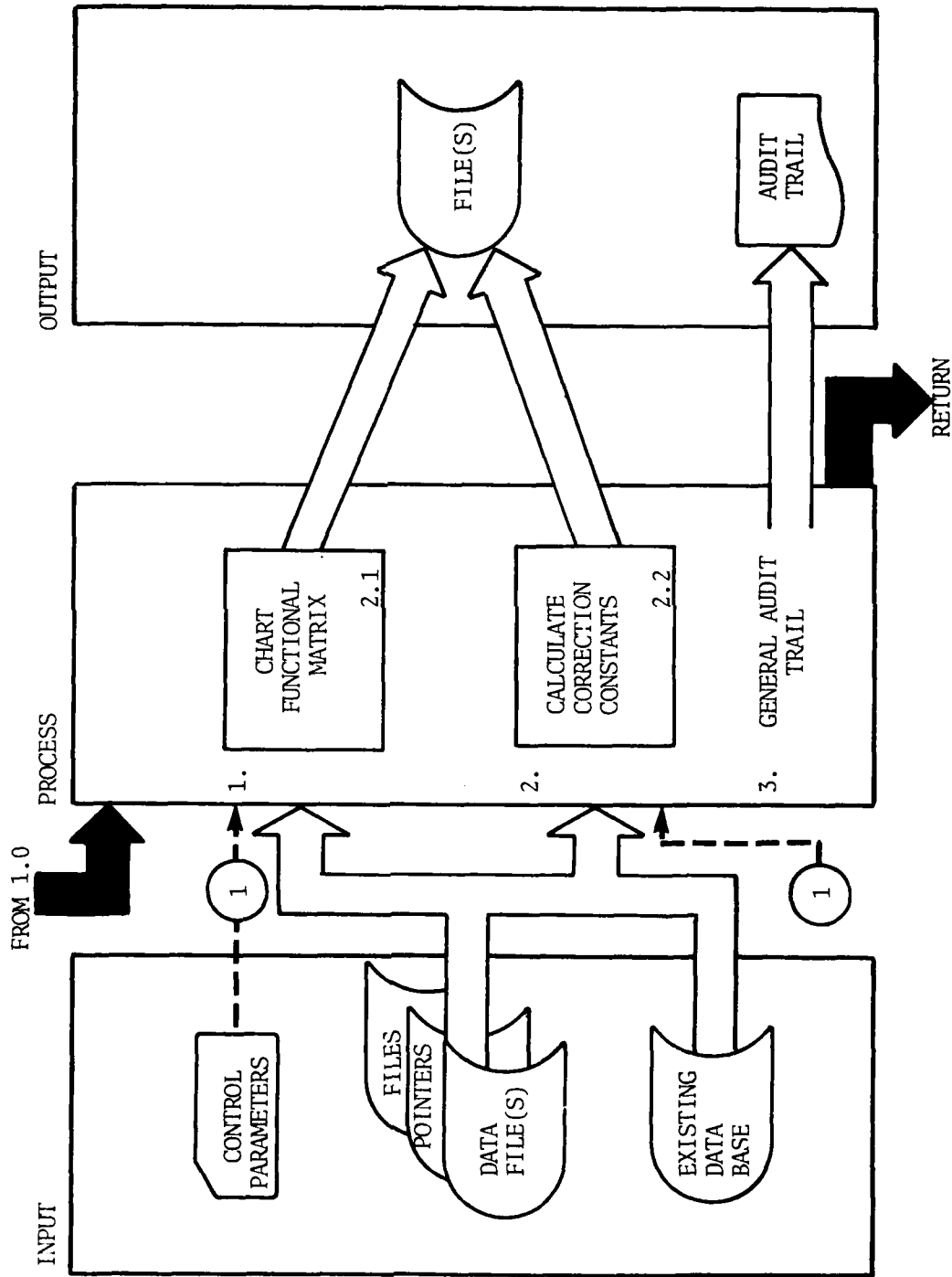


Figure 1. HIPO Chart

- Librarian - assembles, compiles, and link-edits the programs submitted by project programmers. The librarian is responsible for maintaining the library, including work books, record books, subroutines and functions, and configuration control for all source code not maintained by the program support library.

There was a tester assigned to the LSDB project who was not formally a member of the chief programmer team. Nevertheless, he attended all walk-throughs and knew the system almost as well as the project programmers. His primary responsibility was to test the generated code for accuracy and proper function. When a problem arose, he would send an 'Action Item' to the chief programmer regarding the test results.

1.2.4 Structured coding - is based on the mathematically proven Structure Theorem (Mills, 1975) which holds that any proper program (a program with one entry and one exit) is equivalent to a program that contains as control structures only:

- Sequence - two or more operations in direct succession.
- Selection - a conditional branching of control flow. Selection control structures are:
  1. IF-THEN-ELSE
  2. CASE
- Repetition - a conditional repetition of operations while a condition is true, or until a condition becomes true. Repetition control structures are:
  1. DO WHILE
  2. REPEAT UNTIL

These control structures are illustrated in Figure 2. The implementation of these constructs into a computer language allows the implementation of a simpler, more visible control flow which results in more easily understood programs (Dijkstra, 1972).

1.2.5 Structured walk-throughs - A structured walk-through is a review of a developer's work (program design, code, documentation, etc.) by fellow project members invited by the developer. Not only can these reviews locate errors earlier in the development cycle, but reviewers are exposed to other design and coding strategies. A typical walk-



Sequence

BEGIN  
 [statements]  
 END



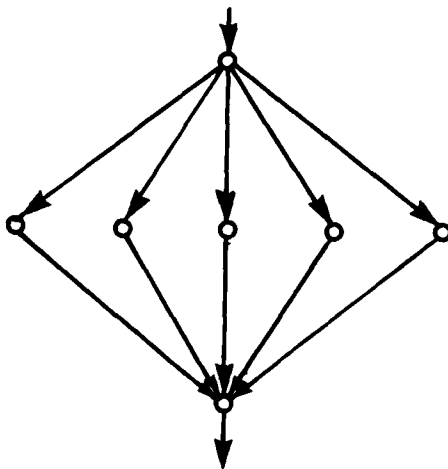
Repetition

WHILE [logical exp.] DO  
 [statements]  
 ENDDO



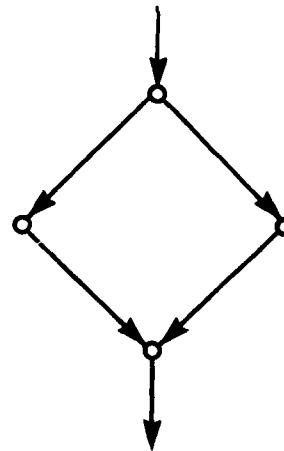
Repetition

REPEAT  
 [statements]  
 UNTIL [logical exp.]



Selection

CASE [expression] OF  
 C<sub>1</sub>: [statement]  
 :  
 C<sub>n</sub>: [statement]  
 END



Selection

IF [logical expression] THEN  
 [statements]  
 ELSE  
 [statements]  
 ENDIF

Figure 2. Control Structures allowed in structured programming

through is scheduled for one or two hours. If the objectives have not been met by the end of the session, another walk-through is scheduled.

During a walk-through reviewers are requested to comment on the completeness, accuracy, and general quality of the work presented. Major concerns are expressed and identified as areas for potential follow-up. The developer then gives a brief tutorial overview of his work. He next walks the reviewers through his work step-by-step, simulating the function under investigation. He attempts to take the reviewers through the material in enough detail to satisfy the major concerns expressed earlier in the meeting, although new concerns may arise.

Immediately after the meeting, the appointed moderator distributes copies of the action list to all attendees. It is the responsibility of the developer to ensure that the points of concern on the action list are successfully resolved and reviewers are notified of the actions taken. It is important that walk-through criticism focus on error detection rather than fault finding in order to promote a readiness to allow public analysis of a programmer's work.

1.2.6 Program support library - The Applied Data Research (ADR) LIBRARIAN software was chosen as the program support library for the LSDB project. A major reason for this choice was its versatility of tools for the IBM 360/65 system. A program support library provides a vehicle for the organization and control of a programming project, the communications among development personnel, and the interface between programming personnel and the computer system. When used with top-down structured programming, the program support library maintains a repository of data necessary for the orderly development of computer programs.

The ADR LIBRARIAN generates a weekly subroutine report which indicates the subroutine size, number of updates, number of runs, etc., during the preceding report period. It is a source program retrieval and maintenance system designed to eliminate problems involved in writing, maintaining, testing, and documenting computer programs. Source programs, test data, job control statements, and any other information normally stored on cards was stored by the ADR LIBRARIAN on tape or disk and updated by a set of simple commands. Data is also stored in hardcopy form in project notebooks. The ADR LIBRARIAN includes the necessary computer and office procedures for controlling and manipulating this data.



### 1.3 Metric Integrated Processing System (MIPS)

MIPS was designed to support SAMTEC operations as the primary source of metric (positional) data processing associated with missile, aircraft, satellite testing, or trajectory measurement activities. The critical function of MIPS is to determine range safety information before launch and establish controls for any potential flight path or abort problems. In preparation for each test, MIPS processes data related to preflight planning, mission parameters, safety parameters, weather data, and instrumentation testing. It provides the capability for real-time missile flight control (safety), post-test data reduction, and metric data analysis. MIPS is capable of accepting metric data input, processing it, and outputting data in appropriate display media for both single and salvo launches.

MIPS was designed around a modular and common data base concept, which together with the control function provides the capability to use the system in a highly structured, yet flexible manner. MIPS was designed to operate with the IBM OS/MVT 360 operating system, and to provide both speed and flexibility in real-time and non real-time processing.

The MIPS development effort consisted of reprogramming and integrating several programs which performed similar functions less flexibly and used the same hardware. The development environments of the projects for separate components were similar since they shared the same management, were implemented on the same computer (IBM 360/65), and were of approximately the same size and duration.

A decision was made at the start of the MIPS project to provide for an evaluation of a highly disciplined programming environment. Two increments of the MIPS project were selected for increased attention and measurement; the Data Analysis Processor using conventional programming techniques and the Launch Support Data Base implemented under the ASTROS plan. The arrangement allowed a quasi-experimental evaluation of the modern programming practices specified in the ASTROS plan.

1.3.1 Launch Support Data Base (LSDB). The LSDB Computer Programming Configuration Item (CPCI) is a non-real-time increment of the MIPS system which determines in advance the range characteristics and possible trajectories of missiles. LSDB represents the reprogramming of an earlier system (VIPARS) which was not referenced during the development. Since LSDB was a redevelopment, the customers were knowledgeable of its function before it was

implemented. LSDB includes both data management functions and complex scientific calculations which are run in batch prior to launch operations without real-time constraints. Specifically, this system provides the processing necessary to generate the Launch Support Data Base parameters used by the Missile Flight Control (MFC) CPCI and by Backup Impact Prediction Software (BIPS) CPCI to support specific launch operations. LSDB also generates the user requested reports, data set files, IBM 7094 GERTS, and the tape files used by the MFC and BIPS from these parameters.

LSDB is composed of five major Computer Programming Components (to be described hereafter as subsystems), each with numerous subroutines and procedures. The five functions performed by subsystems within LSDB are Launch Support Data Base Initialization (LDI), Launch Support Data Base Generation (LDG), Launch Support Data Base Summary Output (LSO), BIPS Launch Support Data Base Preparation (BDP), and Summary Report (BDR). LSDB was developed under the guidelines of the ASTROS system.

1.3.2 Data Analysis Processor (DAP). The DAP reprogramming effort was a sister project to LSDB. DAP analyzed data using parameters generated by LSDB and developed reports for the MIPS system. While the ASTROS plan was not implemented on the DAP project, the management controls and development environment were made as identical as possible to those employed in the LSDB project to provide more valid comparisons between them. The DAP project was implemented in standard FORTRAN, in a non-structured coding environment, by a group of programmers not organized into a chief programmer team.

#### 1.4 Project Environment

1.4.1 Hardware - The processor used in the LSDB project was an IBM 360/65 (768K, 2Mbytes LCS) which was chosen for its numerous and versatile tools. The developmental system was identical to the target system, requiring no conversion effort. The system was available for remote batch and batch use. Turnaround time for batch work was approximately 24 hours and for remote batch was approximately 2 hours. The operating system was IBM OS/360, MVT, release 21 with HASP.

1.4.2 Software - After careful evaluation, S-Fortran was chosen for use on this project. Most of the LSDB code was written in S-Fortran. Small segments were coded in the BAL assembly language. S-Fortran is a high level language (Caine, Farber, and Gordon, 1974) which allows programmers to use structured concepts in their code (Dijkstra, 1972). S-Fortran did allow what Dijkstra (1972) would consider

unstructured constructs. For instance, the UNDO statement allowed exits from loops. A structured precompiler converted the S-Fortran code to standard Fortran for compilation by the standard ANSI-Fortran compiler. The LSDB project had access to a subroutine library for some of the routines needed.

1.4.3 Training - The project personnel underwent a series of courses designed to provide the training necessary to implement the advanced programming techniques specified in ASTROS. The trainer had studied with Yourdon, Inc. prior to teaching these courses. The curriculum included:

- Overview - a survey of the general idea of modern programming practices. (2 hours)
- Structured design - discussions of top-down design, design languages, HIPO charts, threads, and top-down testing (40 hours)
- Structured coding - began with brief discussions of the theory of structured constructs, their history, and mathematical proofs. Primary instruction concerned the use of S-FORTRAN with emphasis on coding and actual problems. (20 hours)
- Program support library - description of the optional features of the ADR LIBRARIAN and their use, including a discussion of the system management facilities. (20 hours)
- Measurement reporting - description of the measurement forms and how to fill them out. Emphasis was given to error classifications and their meanings. (4 hours)
- Management of structured project - stressed the systems management aspects of structured programming, including military standards, measurement reporting and management controls, software life cycle management, chief programmer teams, and structured walk-throughs. (20 hours)

#### 1.5 Data Acquisition

Three types of data were collected on the LSDB project: environment data, personnel data, and project development data. Environment data provided information regarding the system such as the processor, estimated costs, and the amount of source code. Personnel data was information about the people working on the project and their evaluations of

different aspects of it. In order to ensure privacy, strict anonymity was maintained and no evaluation of personnel qualifications was made. Project development data was information describing the development and progress of the project. This information included run reports, manpower loadings, and development information. Table 1 describes these sources of information, and Appendix A contains the data collections forms used during the LSDB project.

RADC has collected development data over a large number of systems, including military and commercial software projects (Duvall, 1978; Nelson, 1978). These data were collected in an attempt to establish baselines and parameters typical of the software development process. Some of the variables against which the LSDB and DAP projects can be compared are listed in Table 2. Appendix B shows the bivariate scatterplots for some of these data.

Table 1  
Data Collection Forms

<u>Instrument</u>	<u>Description</u>
<u>Environment data</u>	
General contract/ project summary	Initial estimates of system characteristics and scope such as total cost and project environment
Management methodology summary	Initial assessments of system reports, schedules, management tools, and standards
Design and processor summary	Initial assessment of the hardware and software configuration of the system
Testing summary	Reported test document preparation and tools, procedures, and visibility mechanisms for test (unavailable)
General wrap-up report	Final statement of project effort and system requirements which allowed a comparison with original estimates in earlier summaries
Source code	The delivered product of the development project
<u>Personnel data</u>	
Personnel profile	Biographical information from each team member on experience, training, and current position
Technology critique	Each team member's evaluation of the tools and techniques used, the training received, and possible future improvements
<u>Project development data</u>	
Action reports	Generated whenever a problem or extraordinary event occurred during development typically requiring program or requirement changes

Computer program run analysis reports	Generated after each computer run and reporting data, computer time, work category, status, and number of statements changed
Computer program failure analysis report	Generated each time a computer run error occurred and reporting data error category and severity
Manpower loadings	Breakdowns of manhours by work category and month
Schedule information	Chronological data on project phases and milestones
Post-development error reports	Generated for each problem or error detected during post-development subsystem and system integration testing which required corrective action

Table 2  
Variables in the RADC Database

Variable	Definition
Program Size	The total number of lines of source code in the delivered product. This count includes declarations, internal program data, and comment lines. It does not include throwaway or external data.
Project Effort	The number of man-months required to produce the software product, including management, design, test, and documentation.
Project duration	The number of months elapsed during the development phase minus dead time such as work stoppages.
Errors	The number of formally recorded software problem reports for which a correction was made during the period covered by the project. This does not include errors from the development portion of the project, but rather from testing through integration.
Derived parameters	Ratios obtained from other variables: <ul style="list-style-type: none"> <li>a. <math>\text{Productivity} = \text{Size}/\text{Effort}</math></li> <li>b. <math>\text{Average Number of Personnel} = \text{Effort}/\text{Duration}</math></li> <li>c. <math>\text{Error Rate} = \text{Errors}/\text{Size}</math></li> </ul>

## 2. THEORIES RELEVANT TO SOFTWARE DEVELOPMENT

### 2.1 Putnam's Software Life Cycle Model

Lawrence Putnam (1978) has refined and verified a software cost and risk estimation model based in part upon the work of Norden (1977). Putnam's model interrelates manpower, effort, development time, level of technology, and the total number of source statements in a system. Putnam's equation describes the behavior of software development projects which consist of over 25,000 lines of code. However, it must be calibrated to each software development organization because of different programming environments.

When provided with initial parameters from the software project, Putnam's model can predict the manpower-time tradeoff involved in implementing the system under development. The model can determine the number of man-years required for a project given limited calendar time. The model is also useful for evaluating corrective action needed on an existing project. Putnam's equation for the developmental phases of the project life cycle is:

$$S_s = C_k K_d^{1/3} t_d^{1/3}$$

where:

- $S_s$  = the size of the final system in source statements,
- $C_k$  = the technology constant,
- $K_d$  = the development effort in man-years,
- $t_d$  = the duration of development time in calendar year.

The technology constant reflects factors influencing the development environment such as use of modern programming practices and hardware capabilities. These factors will determine the time and manpower needed for development. Normally this constant was determined for a particular software organization and input to the model in order to provide cost, time, and risk estimation. However, when size and time are known as in the LSDB project, the model can be applied in retrospect to obtain a technology factor. Solving the software life cycle equation for  $C$  (the technology constant) yields:

$$C_k = \frac{S_s}{K_d^{1/3} t_d^{1/3}}$$



Putnam has further calibrated his technology constant through analyses of data on over 100 systems collected by GE, IBM, TRW, and several government agencies. In development environments typical of 10 to 15 years ago, where programming was performed in batch mode and written in assembly language, values for C could be as low as 1,000. Development environments of systems built with a higher order language such as FORTRAN, in batch processing, on one large mainframe saturated with work, and slow turnaround could yield technology constants of around 5,000. Higher values for C occurred in an environment where modern programming practices were implemented with on-line, interactive programming.

## 2.2 Halstead's Software Science

Maurice Halstead (1977) has developed a theory which provides objective estimates of the effort and time required to generate a program, the effort required to understand a program, and the number of bugs in a particular program (Fitzsimmons & Love, 1978). In 1972, Halstead first published his theory of software physics (renamed software science) stating that algorithms have measurable characteristics analogous to physical laws. According to Halstead (1972, 1977), the amount of effort required to generate a program can be calculated from simple counts of the actual code. The calculations are based on four quantities from which Halstead derives the number of mental comparisons required to generate a program; namely, the number of distinct operators and operands and the total frequency of operators and operands. Preliminary tests of the theory reported very high correlations (some greater than .90) between Halstead's metric and such dependent measures as the number of bugs in a program (Cornell & Halstead, 1976; Funami & Halstead, 1975), programming and debugging time (Curtis, Sheppard, & Milliman, 1979; Gordon & Halstead, 1976), and the quality of programs (Bulut & Halstead, 1974; Curtis, Sheppard, Milliman, Borst, & Love, 1979; Elshoff, 1976; Gordon, 1977). Fitzsimmons and Love (1977), Funami and Halstead (1975), and Akiyama (1971) found that Halstead's effort metric was a much better predictor of the number of errors in a program than either the number of program steps or the sum of the decisions and calls.

2.2.1 Volume. Halstead presents a measure of program size which is different from the number of statements in the code. His measure of program volume is also independent of the character set of the language in which the algorithm was implemented. Halstead defines his measure of program volume as:

$$V = (N_1 + N_2) \log_2 (n_1 + n_2)$$

where,

$\eta_1$  = number of unique operators,

$\eta_2$  = number of unique operands,

$N_1$  = total frequency of operators,

$N_2$  = total frequency of operands.

Halstead's measure of volume is stated in information theoretic terms (Shannon, 1948). He represents volume as the minimum length in bits of all the unique elements in a program (its vocabulary) times the frequency with which these elements are invoked.

2.2.2 Level. Halstead's theory also generates a measure of program level which indicates the power of a language. As the program level approaches 1, the statement of the problem or its solution becomes more succinct. As the program level approaches 0, the statement of a problem or its solution becomes increasingly bulky, requiring many operators and operands. A higher level language is assumed to have more operators available, but these operators are more powerful and fewer operators need to be used to implement a particular algorithm. Halstead's estimate of program level is computed as:

$$\hat{L} = 2\eta_2/\eta_1 N_2.$$

2.2.3 Effort. Halstead theorized that the effort required to generate a program would be a ratio of the program's volume to its level. He proposed this measure as representing the number of mental discriminations a programmer would need to make in developing the program. Halstead's effort metric (E) can be computed precisely from a program (Ottenstein, 1976) which accepts source code as input. The computational formula is:

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

The time required to generate the program can be estimated by dividing E by the Stroud (1966) number of 18 mental discriminations per second.

2.2.4 Scope. In using Halstead's equations to compute the effort or time required to develop a system, it is important to limit the computations to the scope of the program that a programmer may be dealing with at one time. By scope, we mean the portion of a program that a programmer is

attempting to represent to himself cognitively, regardless of whether he is developing code or attempting to understand an existing program. There are several strategies that a programmer could follow while working on a module, and they result in different values for the overall effort.

2.2.4.1 Minimum scope case. In the minimum scope case, the programmer would only keep the context of the subroutine he was currently working on in mind. He would not keep track of other variables from other routines. In this case, each subroutine could be considered a separate program, and the effort for the subsystem would be the summation of the effort for the separate subroutines. In this case the volume of the  $i$ th subroutine would be:

$$V_i = (N_{1_i} + N_{2_i}) \log_2 (\eta_{1_i} + \eta_{2_i}),$$

and the estimated level of the  $i$ th subroutine would be:

$$\hat{L}_i = (2 \eta_{2_i}) / (\eta_{1_i} N_{2_i})$$

The effort for the  $i$ th subroutine would then be:

$$E_i = V_i / \hat{L}_i,$$

where the subscript  $i$  indicates that the values of the variables are computed only from the operators and operands in the  $i$ th subroutine. The total effort for the subsystem is therefore the summation of the efforts of the  $S$  subroutines of the subsystem:

$$E_{MIN} = \sum_{i=1}^S E_i,$$

where  $E_{MIN}$  is the effort for the subsystem, assuming that the programmer developed each subroutine as an independent program and  $S$  is the total number of subroutines.

2.2.4.2 Maximum scope case. Cases 2 and 3 are concerned with the two extremes where there is overlap between the subroutines (e.g., global variables). In these cases, the programmer is assumed to treat the entire subsystem as one program. He mentally concatenates the subroutines into a subsystem and treats the subsystem as one complete algorithm (a gestalt) where he must keep track of all its aspects at any given time.

2.2.4.2.1 Lower maximum scope. In the lower bound for the maximum scope case, we assume that all the variables in the subroutines of the system are the same. In other words, we assume that the programmer will use the same set of

variables for each subroutine, and therefore, the maximum number of unique operators and operands that he must keep track of will be the maximum used in any one subroutine. The effort of the subsystem will then be:

$$E_{LMAX} = \frac{\eta_{1MAX} \sum_{i=1}^S N_{2i} \left( \sum_{i=1}^S (N_{1i} + N_{2i}) \log_2 (\eta_{1MAX} + \eta_{2MAX}) \right)}{2^{\sum_{i=1}^S \eta_{2i}}}$$

where,

$\eta_{1MAX}$  = the number of operators in the subroutine with the largest number of unique operators,

$\eta_{2MAX}$  = the number of operands in the subroutine with the largest number of unique operands,

$N_{1i}$  = the frequency of operators in subroutine  $i$ ,

$N_{2i}$  = the frequency of operands in subroutine  $i$ ,

$S$  = the number of subroutines in the system.

2.2.4.2.2 Upper Maximum scope. To establish an upper bound for the maximum scope case, we make the assumption that the operators and operands for each separate section of code are treated as being unique to that section of code (local variables appearing in that section and no other). The absolute upper bound for the effort required to program the system is:

$$E_{UMAX} = \frac{\sum_{i=1}^S \eta_{1i} \left( \sum_{i=1}^S N_{2i} \left( \sum_{i=1}^S (N_{1i} + N_{2i}) \log_2 \left( \sum_{i=1}^S (\eta_{1i} + \eta_{2i}) \right) \right) \right)}{2^{\sum_{i=1}^S \eta_{2i}}}$$

A mathematical proof is presented in Appendix C to show that  $E_{MIN} < E_{LMAX} < E_{UMAX}$ .

### 2.2.5 Delivered Errors

Halstead (1977, p. 85) developed a formula for predicting the number of errors during system testing. The equation he presents is  $B = V/E_{CRIT}$ , where  $B$  is the number of

errors expected,  $V$  is the volume, and  $E_{CRIT}$  is "the mean number of elementary discriminations between potential errors in programming" (p. 85).  $E_{CRIT}$  can also be expressed as the critical value for effort computed from a critical value for volume above which, at least one error is predicted to occur.  $E$  is shown to be  $(V^*_{CRIT})^3 / \lambda^2$ , where  $V^*_{CRIT}$  is the minimal value of  $E$  for an amount of code which would consume the available space in the working (short term) memory of the average programmer at a single instant and  $\lambda$  is the level of the implementation language derived from tables developed by Halstead for a number of languages. Based on his assumptions about memory capacity, Halstead derives a value for  $V^*$  of 24. Therefore, the revised formula for the prediction of delivered bugs is:

$$B = V/E_{CRIT} = \frac{V}{24^3/\lambda^2} = \frac{V\lambda^2}{13,824} .$$

### 2.3 McCabe's Complexity Metric

Thomas McCabe (1976) defined complexity in relation to the decision structure of a program. He attempted to assess complexity as it affects the testability and reliability of a module. McCabe's complexity metric,  $v(G)$ , is the classical graph-theory cyclomatic number indicating the number of regions in a graph, or in the current usage, the number of linearly independent control paths comprising a program. The computational formula is:

$$v(G) = \# \text{ edges} - \# \text{ nodes} + 2(\# \text{ connected components}).$$

Simply stated, McCabe's metric counts the number of basic control path segments through a computer program. These are the segments which when combined will generate every possible path through the program. McCabe presents two simpler methods of calculating  $v(G)$ . McCabe's  $v(G)$  can be computed as the number of predicate nodes plus 1, where a predicate node represents a decision point in the program. Thus  $v(G)$  may be most easily envisioned as the number of regions in a planar graph (a graph in regional form) of the control flow. Figure 3 presents several examples of how to compute  $v(G)$  from a planar flowchart of the control logic.

The simplest possible program would have  $v(G) = 1$ . Sequences do not add to the complexity. IF-THEN-ELSE, DO-WHILE, or DO-UNTIL increase the complexity by 1. It is assumed that regardless of the number of times a DO loop is executed there are really only two control paths: the straight path through the DO and the return to the top. Clearly a DO executed 25 times is not 25 times more complex

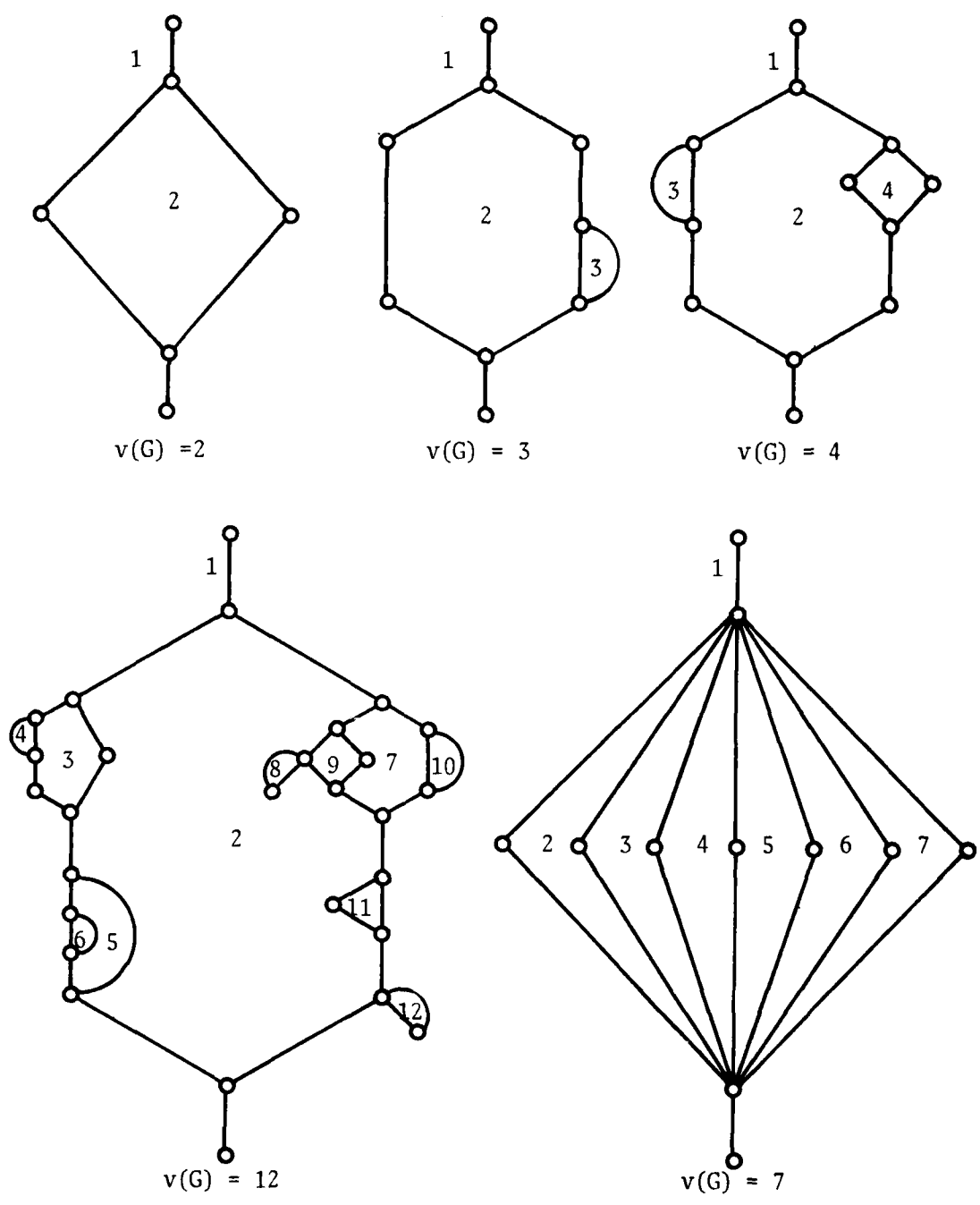


Figure 3. Control flow graphs and associated values for  $v(G)$

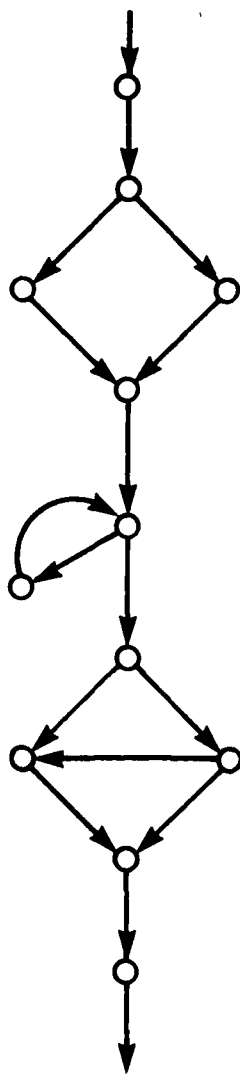
than a DO executed once.

McCabe also presents a method for determining how well the control flow of a program conforms to structured programming practices. He decomposes the control flow by identifying structured constructs (sequence, selection, and repetition) that have one entrance and one exit and replaces them with a node. If a program is perfectly structured, then the flow graph can be reduced to a simple sequence with a complexity of 1 by iteratively replacing structured constructs by nodes. If a segment of code is not structured, then it cannot be decomposed and will contribute to the complexity of the program. Figure 4 demonstrates such an analysis. McCabe calls the ultimate result of this decomposition the essential complexity of the program. The essential complexity of the reduced graphs indicates the unstructured complexity of the system.

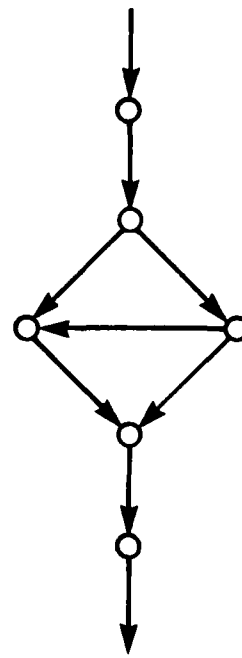
#### 2.4 Software Quality Metrics

One of the most comprehensive studies of software quality was performed by McCall, Richards, and Walters (1977) under an RADC contract. They defined various factors of software quality as a means for quantitative specification and measurement of the quality of a software project. With these aids, a software manager can evaluate the progress and quality of work and initiate necessary corrective action.

A set of quality factors was identified in a literature review. The importance of the factors was determined through a survey of knowledgeable users. From results of this survey a set of software quality factors was chosen (Table 3) and classified as to their impact in the software development process. For each software quality factor, a set of underlying metrics was developed which allowed objective measurement of specific aspects of software which would impact quality. Each metric was chosen as a descriptor of some aspect of the software development process such as simplicity. A subset of the metrics in the McCall et al. study was used in the current study to demonstrate the quality of code produced for LSDB as compared to code from DAP. Appendix D presents the seven metrics selected and their derivation.



Flowgraph  
 $v(G) = 5$



Flowgraph of  
 essential complexity  
 $v(G) = 3$

Figure 4. Decomposition of a control flowgraph to essential complexity



Table 3  
Software Quality Factors

Factor	Definition
Correctness	Extent to which a program satisfies its specifications and fulfills the user's mission objectives.
Reliability	Extent to which a program can be expected to perform its intended function with required precision.
Efficiency	The amount of computing resources and code required by a program to perform a function.
Integrity	Extent to which access to software or data by unauthorized persons can be controlled.
Usability	Effort required to learn, operate, prepare input, and interpret output of a program.
Maintainability	Effort required to locate and fix an error in an operational program.
Testability	Effort required to test a program to insure it performs its intended function.
Flexibility	Effort required to modify an operational program.
Portability	Effort required to transfer a program from one hardware configuration and/or software system environment to another.
Reusability	Extent to which a program can be used in other applications - related to the packaging and scope of the functions that programs perform.
Interoperability	Effort required to couple one system with another.

### 3. DATA ANALYSES

Data analyses will be reported in two sections. The first section reports comparative analyses between LSDB and DAP. The order of presentation will proceed from analyses at the project level to detailed analyses of the source code. The comparisons will include:

1. the descriptive data generated from both projects,
2. the technology levels as determined from Putnam's model,
3. the efforts involved in generating the code as determined from Halstead's theory,
4. the complexity of the control structures of the systems as indexed by McCabe's metric,
5. the quality of the delivered source codes indicated by software quality metrics, and
6. comparison to projects in the RADC database.

The second section of results will present analyses of the error data collected on the LSDB project. These analyses will include:

1. descriptive data for run-error categories,
2. comparison with error categories from TRW data,
3. prediction of run-errors by subroutine,
4. trends in run-errors across time,
5. prediction of post-development errors.

There were some seeming discrepancies in the data. While run-error reports were obtained for the LSDB project over a 16 month period, man-hour loadings were only reported for 14 months. This discrepancy appears even greater in that numerous runs were reported during the requirements and preliminary design phases of the project, although no hours were logged to coding. The LDI component was not actually peculiar to the LSDB system. Thus, coding on LDI began immediately at the start of the LSDB project since its requirements were known and the team wanted practice writing in S-Fortran. As the preliminary design progressed on the remainder of the LSDB components, S-Fortran listings were

used during walk-throughs in place of a Program Design Language. Some of the early man-hours appear to have been charged to an account other than that of LSDB. Thus, results of analyses involving man-hours should be interpreted as approximate rather than exact.

### 3.1 Comparative Analyses: LSDB Versus DAP

3.1.1 Descriptive data. The source code for the LSDB Computer Program Configuration Item contained over three times as many source lines and almost two and one half as many executable lines as did that of DAP (Table 4). Executable lines represented the source code minus comments, data statements, and constructs not strictly applicable to the algorithm. ENDIF, READ, and WRITE statements were not counted as executable lines. The original estimate of 5,000 lines source code in S-FORTRAN for LSDB was close to the number of lines of strictly algorithmic executable code actually produced. The 16,775 source lines in LSDB were distributed across five subsystems (CPC's) which typically ranged from 1700 to 2700 lines, with the exception of one 8013 line subsystem. The six subsystems constituting DAP were all smaller, ranging from 200 to 1400 lines of code.

Comments accounted for a larger percentage of the LSDB source code (38%) than of the DAP code (22%). While there were fewer executable lines than comment lines (31% vs. 38%, respectively) in the LSDB code, there were almost twice as many executable lines as comment lines (44% vs. 23%) in the DAP code.

The LSDB project required approximately 8081 man-hours of effort to complete, while the DAP project required approximately 6782 man-hours. Thus, while LSDB contained 239% more total source lines of code and 140% more executable lines than DAP, the LSDB project required only 19% more man-hours of effort to complete.

Figure 5 presents a plot of the man-hours expended on LSDB and DAP during each month of the project from the requirements phase through development testing and system documentation (Appendix E). Different profiles were observed across the 14 months required for each of the two projects. For LSDB, the largest loadings occurred during the initial five months of the project. With the exception of month 8, only between 300 and 600 man-hours of effort were expended during each of the last nine months of the LSDB project. Man-hours expended on DAP, however, were greatest during the final months of the project. That is, during the initial 8 months of DAP only 150 to 450 man-hours were expended per month, while during the last six months (except for month 13)

Table 4

Number of lines by Subsystem: LSDB versus DAP

Subsystem	Total lines	Non-comment lines	Executable lines
<u>LSDB:</u>			
LDI	1757	1139	636
LDG	2527	1469	833
LSO	1801	1348	775
BDP	8013	4333	2009
BDR	2677	2024	952
Total	16775	10413	5205
<u>DAP:</u>			
1	1357	1030	514
2	951	748	440
3	714	592	375
4	1336	1064	591
5	395	312	184
6	200	114	65
Total	4953	3860	2169

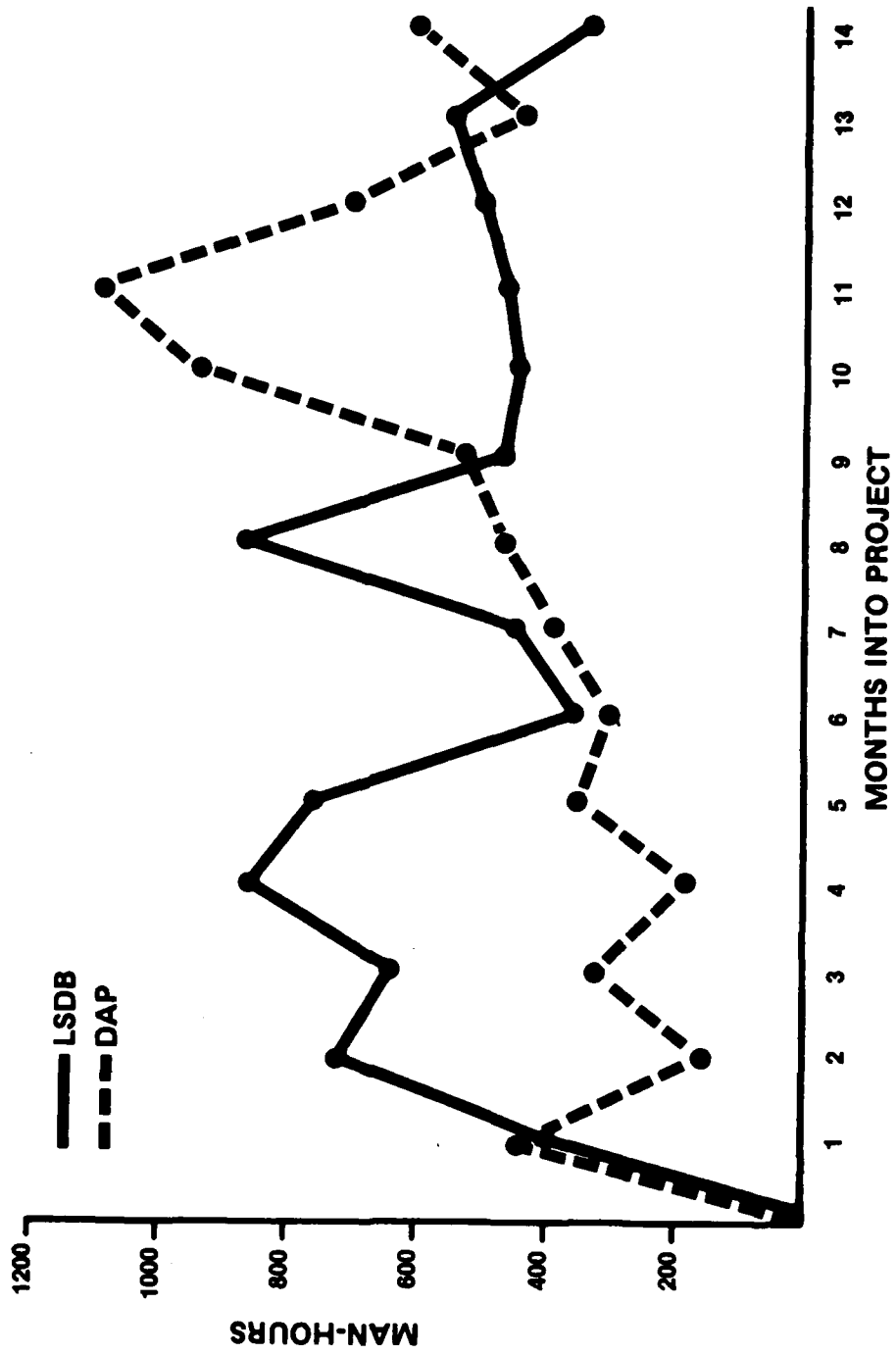


Figure 5. Chronological manpower loading for LSDB and DAP

man-hour expenditures ranged between 500 and 1050. Figure 6 presents a chronological history of both projects by life cycle phase.

The percentages of man-hours expended during each phase of development for LSDB and DAP are presented in Figure 7. On both projects, approximately 20% of the man-hours were expended in coding and integration, while 15% were expended in product documentation and training. Differing profiles of effort expenditure were observed on the other four phases for each project. While only 6% of the man-hours on LSDB were invested in requirements analysis, 21% of those on DAP were consumed during this phase. However, it is likely that many of the man-hours devoted to LSDB requirements analysis may not have appeared among the manpower loadings charged to this project. Almost one third of the requirements hours for DAP were expended in what appears to have been a modification of the requirements during month 10. When compared to the DAP project, the percentage of total man-hours on the LSDB project invested in the preliminary design was almost twice as great, and in the detailed design was over four times as great. One quarter of the man-hours on the LSDB project were expended in development testing and evaluation, while one third of those on the DAP project were so expended. However, half of the testing-related time on the LSDB project was invested in preparation of the test procedures, compared to only an eighth of the DAP test-related time being devoted to preparation.

### 3.1.2 Level of Technology: Putnam's Model

Putnam's technology constant for the LSDB project was anticipated to fall in the midrange of values (approximately 5,000) because advanced tools were employed, but the computer was used in batch mode, turn around time was slow, programming was non-interactive, and some of the code was written in assembler. Putnam argues that in calculating the technology constant for a development effort, calendar time and development effort should be measured only from the date that work on the detailed design was initiated. Further, the constant was computed using the number of non-comment lines. The level of technology for LSDB was calculated as follows:

$$C_k = \frac{10,413 \text{ lines of code}}{(3.78\text{M/y})^{1/3} (1 \text{ calendar yr.})^{1/3}} = 6,685$$

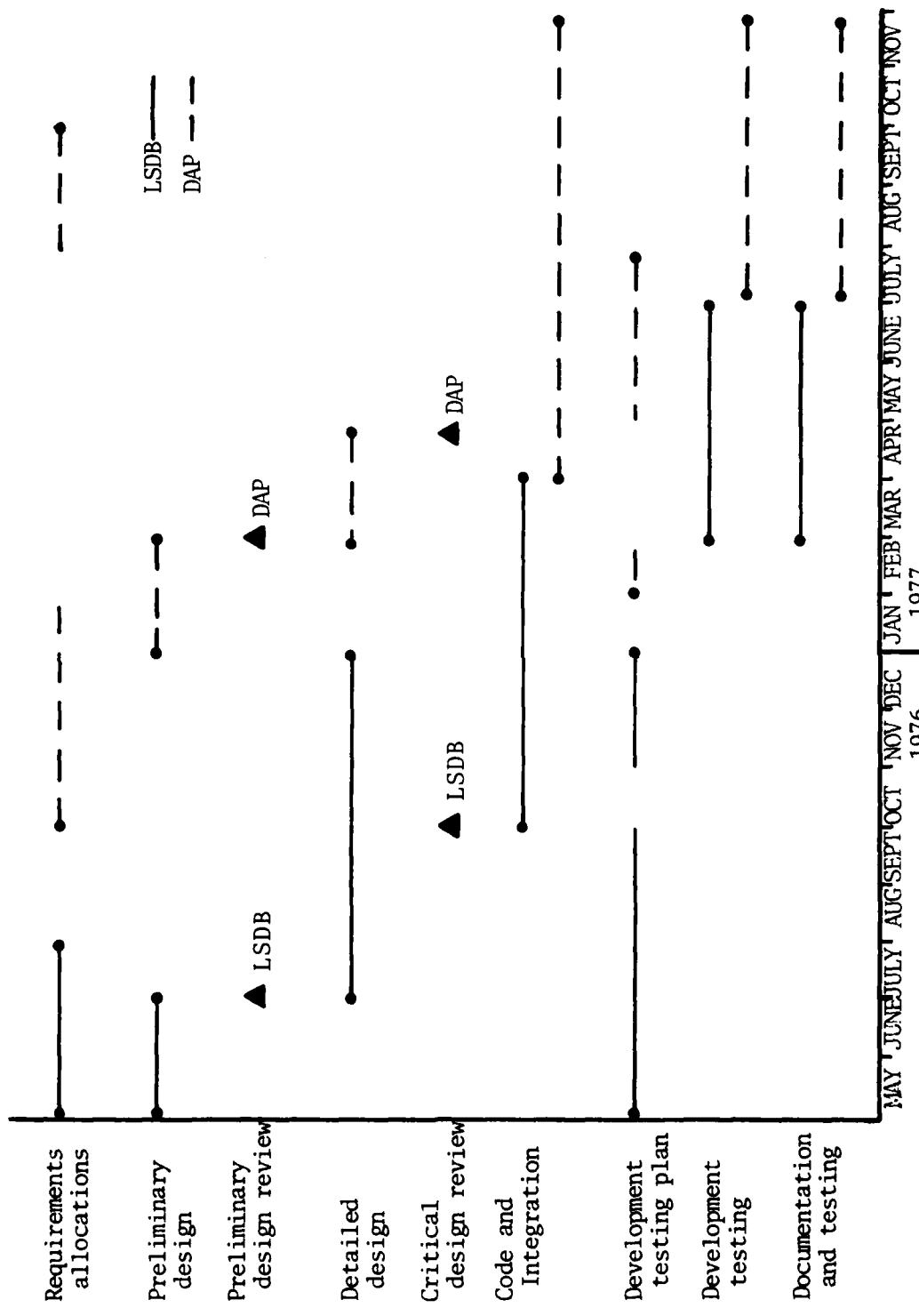


Figure 6. Chronological history of LSDB and DAP by phase

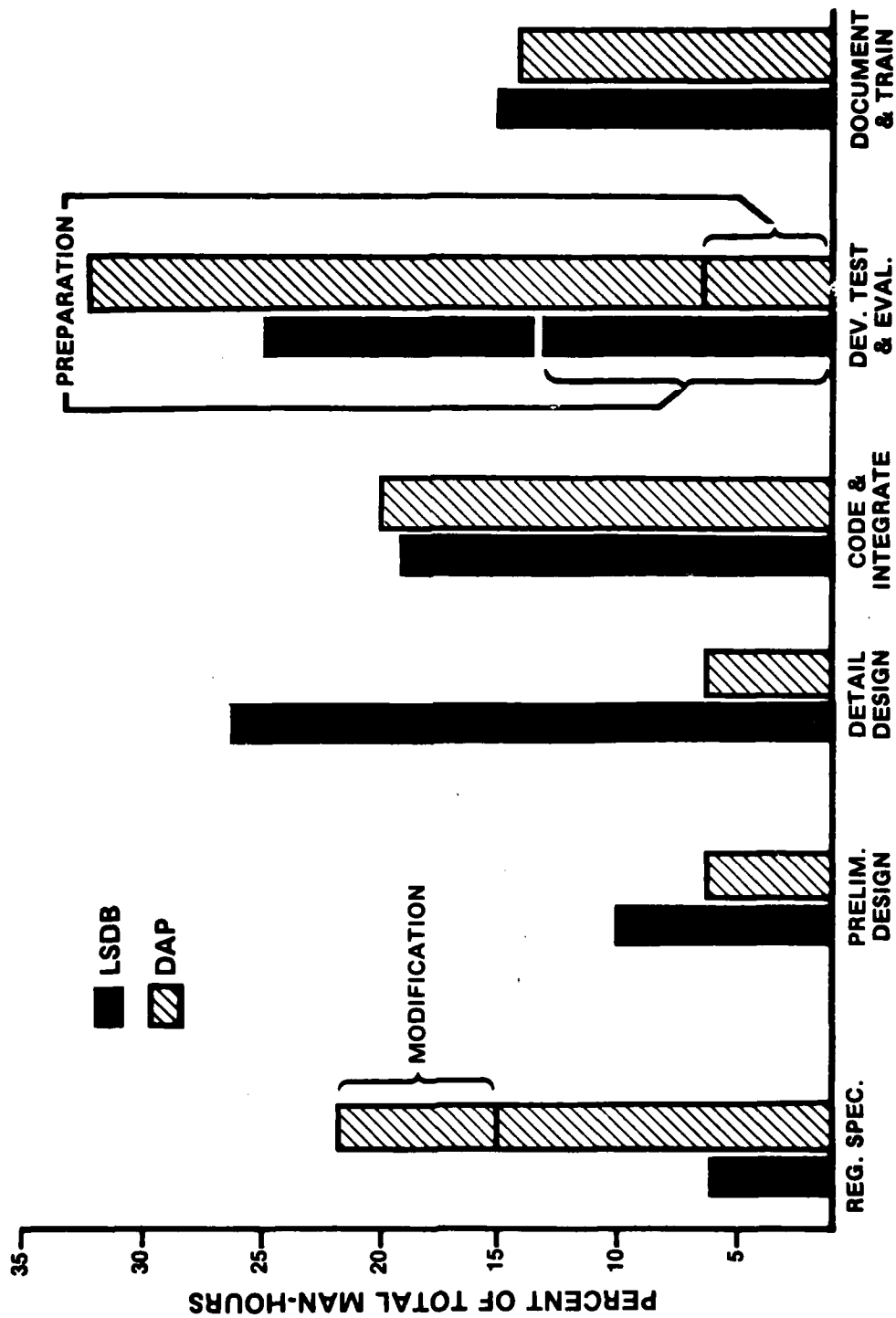


Figure 7. Percentage of effort by development phase



The level of technology for DAP was computed as follows:

$$C_k = \frac{3,680 \text{ lines of code}}{(2.75M/y)^{1/3} (.75 \text{ calendar yr.})^{1/3}} = 2,891$$

The technology constant computed for DAP is lower than that computed for the LSDB project. Since both projects used the same machine and access times were similar, differences in the technology constant cannot be attributed to a factor Putnam considers to be one of the primary influences on his metric; access time. Although Putnam suggests that his equation will yield an overestimate of the technology constant for projects of under 25,000 lines of code, these computations nevertheless validate the ASTROS plan as providing a more advanced programming technology than the conventional techniques practiced on the DAP project.

### 3.1.3 Programming Scope and Time: Halstead's Model

The volumes, estimated levels, and efforts for each level of scope described in Section 2.2.4 for each of the subsystems from LSDB and DAP are presented in Table 5. The subsystems were treated as separate entities for the purpose of analyses presented here. Thus, the maximum scope cases are calculated on these subsystems separately rather than on the total LSDB system.

Regardless of the level of scope at which the analysis is conducted, the total volume of the LSDB code is greater than that of DAP. However, if we assume that programmers were maintaining a mental representation of an entire subsystem at one time (the two maximum scope cases), the effort needed to create the LSDB subsystems was greater and LSDB's level was lower than those associated with the DAP code. However, if programmers are only attempting to track one subroutine at a time (minimum scope), then the effort for LSDB is less and the level greater than for DAP.

In order to determine which level of scope was most accurately reflected in the data, the efforts associated with each level of scope were used to predict the time required to code each project. For each effort measure, the total number of mental discriminations was divided first by the Stroud number (18 discriminations per second; Stroud, 1966) and then by the 3,600 seconds in an hour to obtain a prediction of the man-hours required for coding. Since the Stroud number represents an optimum measure of mental performance, the effort measure which most accurately reflects the real effort

Table 5

Volume, Level, and Effort by Levels of Scope for Subsystems from Each Project

Subsystem	Minimum scope			Lower maximum scope			Upper maximum scope		
	V <sub>MIN</sub>	$\hat{L}_{MIN}$	E <sub>MIN</sub>	V <sub>LMAX</sub>	$\hat{L}_{LMAX}$	F <sub>LMAX</sub>	V <sub>UMAX</sub>	$\hat{L}_{UMAX}$	E <sub>UMAX</sub>
<u>LSDB</u>									
LDI	33100	.0058	5.70M	33915	.0021	16.15M	40812	.0019	21.48M
LDG	44400	.0079	8.24M	45558	.0015	30.37M	57862	.0015	38.57M
LSO	29800	.0076	4.80M	31524	.0021	15.01M	38631	.0019	20.33M
BDP	99450	.0093	13.75M	111037	.0013	85.41M	144160	.0009	160.17M
BDR	43043	.0084	4.44M	47796	.0035	13.66M	55047	.0019	28.97M
Total	249793	.0082	36.93M	269830	.0019	160.60M	336512	.0014	269.52M
<u>DAP</u>									
1	34540	.0040	17.83M	36100	.0013	27.77M	39933	.0011	36.30M
2	22960	.0070	11.04M	24306	.0017	14.30M	25943	.0017	15.26M
3	24434	.0018	13.57M	24434	.0018	13.57M	24434	.0018	13.57M
4	34700	.0032	26.69M	36475	.0009	47.53M	39367	.0009	43.74M
5	11422	.0042	2.72M	11422	.0042	2.72M	11422	.0042	2.72M
6	3531	.0302	.12M	3531	.0302	.12M	3531	.0302	.12M
Total	131587	.0046	71.97M	136268	.0023	49.01M	144630	.0022	111.71M

Note: Values of  $\hat{L}_{MIN}$  represents weighted means of values for level in separate subroutines.

involved in developing the code should underestimate the actual coding time. This underestimation allows for the fact that programmers do not work at peak efficiency throughout their coding hours.

As evident in Figure 8, the two maximum scope cases overestimate the actual coding time for both LSDB and DAP. This overestimation is substantial for LSDB. Thus, the minimum scope case where programmers only attempt to keep track of the subroutine they are currently working on appears to be best represented in the data from both projects. The substantial underestimation of the coding time on the LSDB project agrees with the reports of project programmers that they had considerable slow time during the coding phase due to poor turnaround time. It is also likely that the scope of the program that programmers were working with at a given time was somewhat larger than described by the assumptions of the minimum scope case (e.g., programmers may have to remember several global variables).

Since the data provide the best support for the assumptions underlying the minimum scope case, it appears that the effort required to code LSDB was less than that required to program DAP. This may have occurred in part because the programming level evident in the source code was greater for LSDB when the assumptions of the minimum scope case are accepted. That is, the LSDB code appears to have been written more succinctly than the DAP code.

This comparison becomes even more evident if we use the productivity defined as non-comment lines produced per man-hour from each project to predict the programming time for the other project. As evident in Figure 8, if the rate of non-comment lines produced per man-hour that was observed on the DAP project had been true for the LSDB project, it would have taken almost two and one-half times as many man-hours to produce the non-comment portion of the LSDB code. Similarly, had the production rate of the LSDB project been achieved on the DAP project, the DAP code could have been completed much more quickly. Thus, the LSDB code was produced with substantially less effort than was the DAP code.

#### 3.1.4 Complexity of Control Flow: McCabe's Model

In order to evaluate the complexity of the control flows using McCabe's procedures, two subroutines were selected for analysis from the source code of each project. McCabe's  $v(G)$  was computed on each subroutine and the values ranged from 76 to 165 (Table 6, Column 1). However, differences in the number of executable statements in each subroutine over which

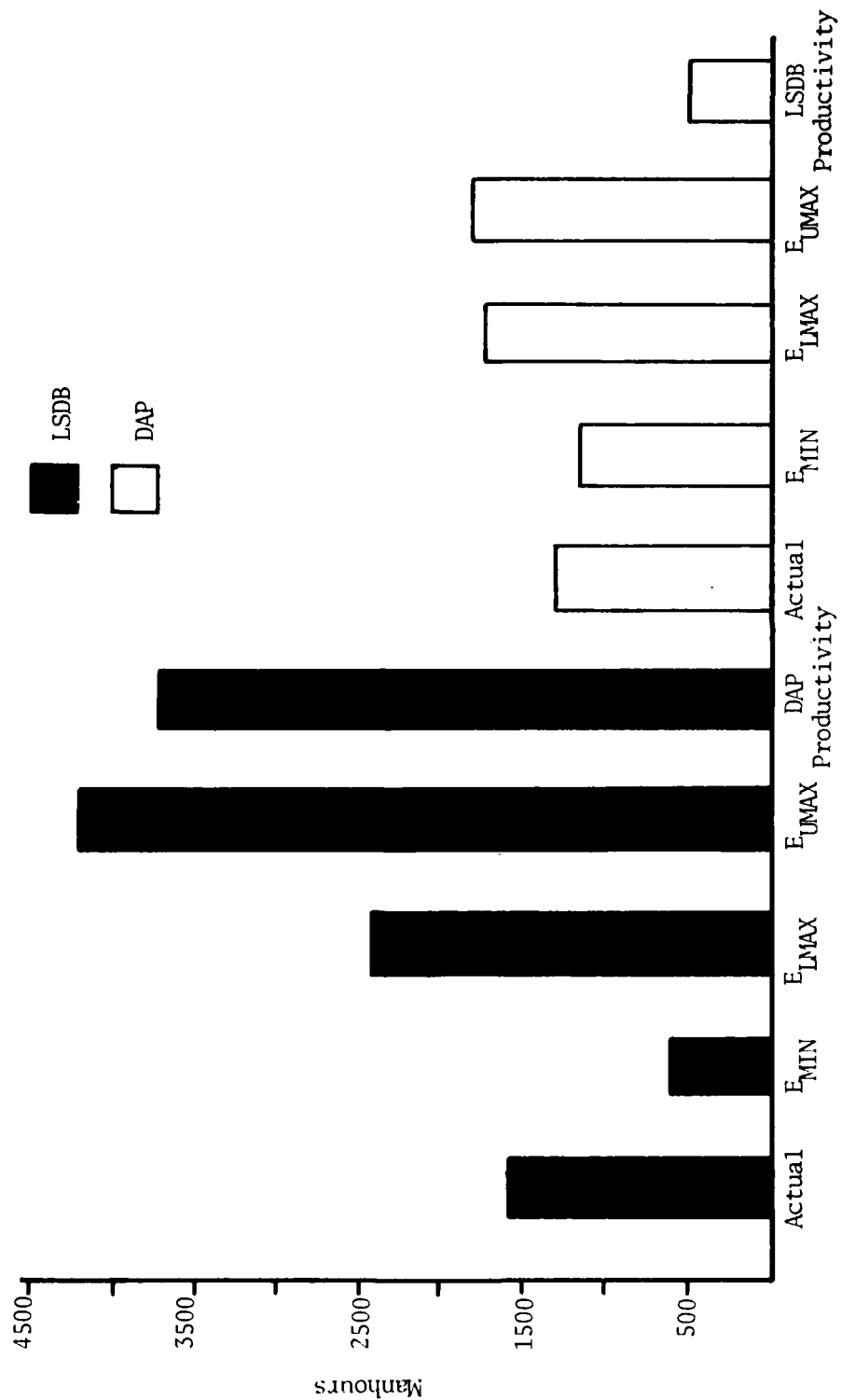


Figure 8. Predictions from Halstead's model of time to code both projects

Table 6

McCabe Values for Selected Subroutines from Each Project

Subroutine	Raw data		Adjusted values	
	v(G)	Essential v(G)	v(G)	Essential v(G)
LSDB:				
1	76	31	101	41
2	136	18	119	18
DAP:				
1	147	50	114	39
2	165	56	132	45

these values were computed made their comparison difficult. Values of  $v(G)$  for each subroutine were adjusted proportionately to a value for a standard subroutine of 300 executable statements. That is, the ratio of  $v(G)$  to lines of code for each subroutine was calculated, and then was standardized to a baseline of 300 lines of code. Following this transformation, the control flows of the subroutines from each project were found to be of approximately equal complexity (Table 6, Column 3).

The degree to which the code from each module exhibited a control structure consistent with the structured principles of Dijkstra (1972) was also evaluated using McCabe's (1976) procedures for establishing essential complexity. Each control structure consistent with those allowed by Dijkstra was replaced in a flowchart by a node until all such structures had been replaced. The result of this replacement process for perfectly structured programs would be a sequence with essential  $v(G) = 1$ , and values greater than 1 would indicate the extent of "unstructuredness" in the control flow. The essential complexity of the four modules ranged from 31 to 56. The adjusted scores indicated that one of the subroutines from LSDB was similar to those from DAP in its degree of unstructuredness, while the other was substantially more structured. Although this unstructuredness in LSDB seems surprising, S-FORTRAN allows an UNDO statement, which results in an unstructured construct by allowing jumps out of loops (Figure 9). With the exception of this construct, the LSDB code in the two modules analyzed was consistent with the principles for structured code described by Dijkstra. The violations of structured control flow in the DAP code were much more varied.

### 3.1.5 Software Quality Metrics

Several software quality metrics concerning modularity, simplicity, and descriptiveness reported by McCall, Richards, and Walters (1977) were computed on the subroutines used in the McCabe analysis. The detailed scoring of the software quality metrics relevant to this study can be found in Appendix D. Table 7 summarizes the results of this analysis. On two of the six measures studied the LSDB code was found to be clearly superior to the DAP code. That is, LSDB was written in a language which incorporated structured constructs, and the system design was generally implemented in a modular fashion. The LSDB code in the subroutines analyzed deviated from the principles of top-down modular design to the extent that calling procedures did not define controlling parameters, control the input data, or receive output data. The LSDB subroutines received slightly higher scores for coding simplicity than those of DAP. These scores

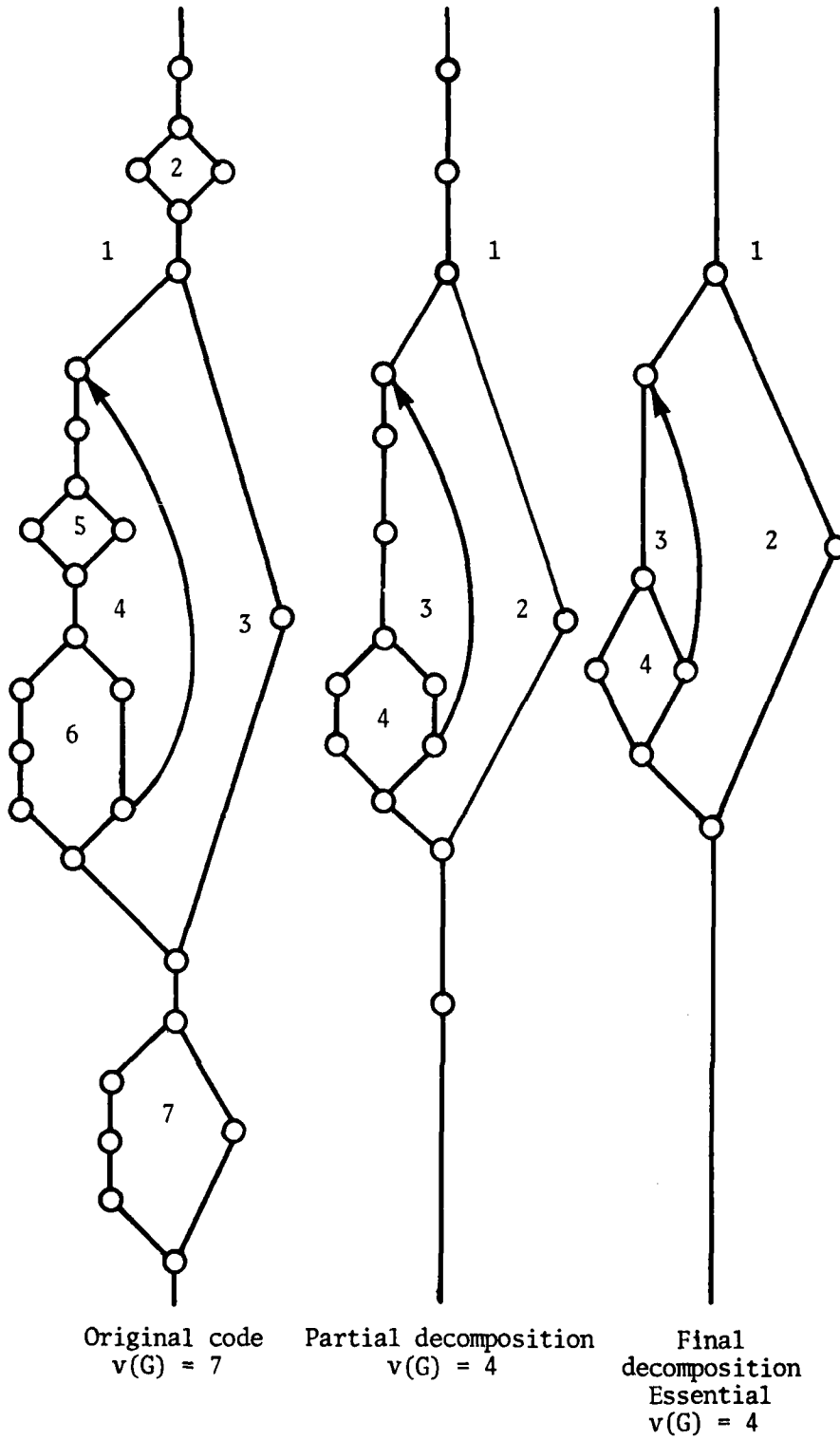


Figure 9. An example from LSDB of decomposition to essential complexity

Table 7

## Values of Software Quality Metrics for Selected Subroutines

Metric	LSDB		DAP	
	1	2	1	2
Structured language	1.00	1.00	.00	.00
Coding simplicity	.67	.69	.63	.57
Modular implementation	.55	.41	.00	.00
Quantity of comments	.49	.28	.46	.17
Effectiveness of comments	.45	.45	.43	.40
Descriptiveness of implementation language	1.00	.98	.83	.83



reflected better implementation in LSDB of top to bottom modular flow, use of statement labels, and avoidance of GO TO's. Minor differences were observed between projects on the effectiveness of comments. Thus, the greater percentage of comments in the LSDB code may not have contributed proportionately more to the quality of documentation beyond the quality observed in DAP. The descriptiveness of the LSDB code (e.g., use of mnemonic variable names, indentation, single statement per line, etc.) was slightly greater than that of the DAP code.

### 3.1.6. Comparison to RADC Database

Richard Nelson (1978) has performed linear regressions of delivered source lines of code on other variables in the RADC software development database. These regressions allow an investigation of performance trends as a function of project size. When outcomes from the LSDB and DAP projects are plotted into these regressions, it becomes possible to compare the performance of LSDB and DAP with other software development efforts while controlling for the size of the project (in terms of delivered source lines).

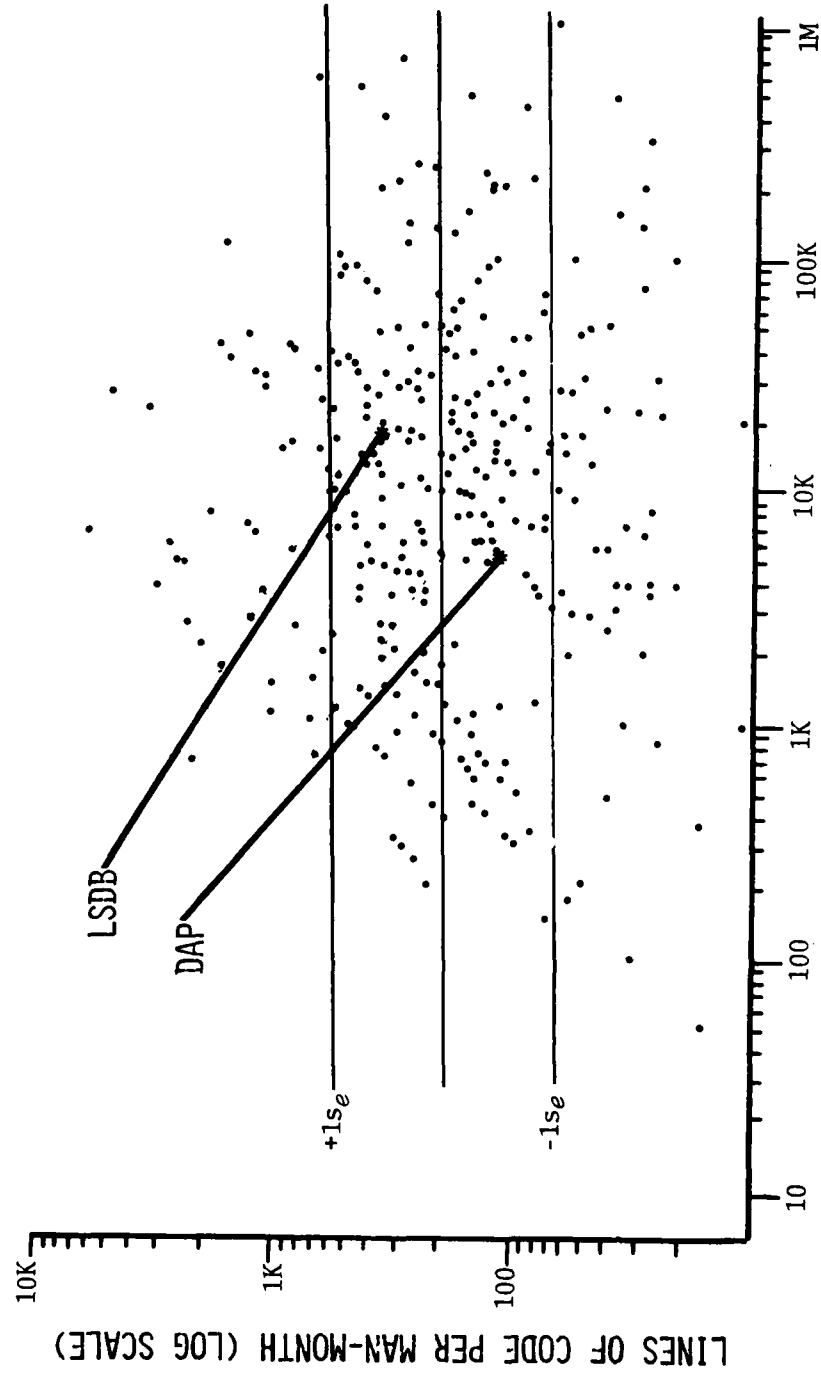
Figure 10 presents the scatterplot for the regression of delivered source lines of code on productivity (lines of code per man-month). The datapoints for the LSDB and DAP projects fall within one standard error of estimate of the regression line. However, LSDB falls above the regression line and DAP falls below it, suggesting that LSDB's productivity was slightly higher than the average productivity for projects of similar size and DAP's was slightly lower. Scatterplots presented in Appendix B for regressions of delivered source lines on total man-months, project duration, total errors, error rate, and number of project members indicated similar results. That is, the performance of LSDB was usually better than that of DAP when adjusted for differences in the number of lines of code. However, the performance of the LSDB project was only slightly better than average when compared against results of other projects.

## 3.2 Error Analyses

All of the data reported in this section are from the LSDB project with the exception of the post-development errors for which data were available from DAP. No records of development runs were available from DAP.

### 3.2.1 Error Categories

3.2.1.1 Descriptive data. Of the 2,719 computer



DELIVERED SOURCE LINES OF CODE (LOG SCALE)

Figure 10. Regression of delivered source lines of code on productivity

runs involved in the development of LSDB, 508 (19%) involved an error of some type (see Appendix F). The frequencies of these errors are reported in Table 8 by a categorization scheme similar to that developed by Thayer et al. (1976) for TRW Project 5. The 14 categories of errors are divided into two groups and a lone category of unclassified errors. The first group involved errors which were detected by a malfunction of the algorithm implemented in the code. These categories included computational errors, logic errors, data errors, etc., and accounted for 28% of the total errors recorded. Most numerous were logic errors which constituted 19% of all errors.

Over half of the errors recorded (55%) were non-algorithmic and involved the hardware functioning (23%), the job control language (14%), keypunch inaccuracies (10%), or program execution errors (i.e., compile error or execution limits exceeded, 8%). Seventeen percent of the errors recorded could not be classified into an existing category in either group.

3.2.1.2 Comparison with TRW data. As a partial test of the generalizability of these error data, the profile across selected error categories was compared to similar data from three development projects conducted at TRW (Thayer et al., 1976). Data are reported only for those errors for which similar classifications could be established. This analysis was performed and first reported by Hecht, Sturm, and Trattner (1977). The percentages reported in Figure 11 were developed by dividing the number of errors in each category by the total number of errors across the five categories. LSDB and the fourth study from the TRW data were found to be quite similar, especially with regard to the low percentage of computational errors and the high percentage of logic errors. The LSDB error profile was similar to the other two TRW studies in the percent of data input and handling errors and interface/program execution errors. Overall, it would appear that the distribution of error types on the LSDB project is similar to distributions observed on other projects.

### 3.2.2 Prediction of Errors at the Subsystem Level

Table 9 presents the total number of runs involved in developing each subsystem of LSDB. These runs are further broken down by the number that contained an error of any type and the number that contained an algorithmic error. Nineteen percent of the runs contained some type of error, while 5% contained an algorithmic error. There were 718 runs that were not classified as belonging to a particular subsystem. These runs appear to have involved either subsystem

Table 8

## Frequencies of Error Categories

Error category	Freq.	%
<u>Algorithmic errors:</u>		
Computational	5	1
Logic	98	19
Data input	17	3
Data Handling	12	3
Data output	3	1
Interface	0	0
Array processing	1	0
Data base	4	1
Total	140	28
<u>Non-algorithmic errors:</u>		
Operation	115	23
Program execution	41	8
Documentation	0	0
Keypunch	51	10
Job control language	73	14
Total	280	55
<u>Unclassified:</u>	88	17
Grand total	508	

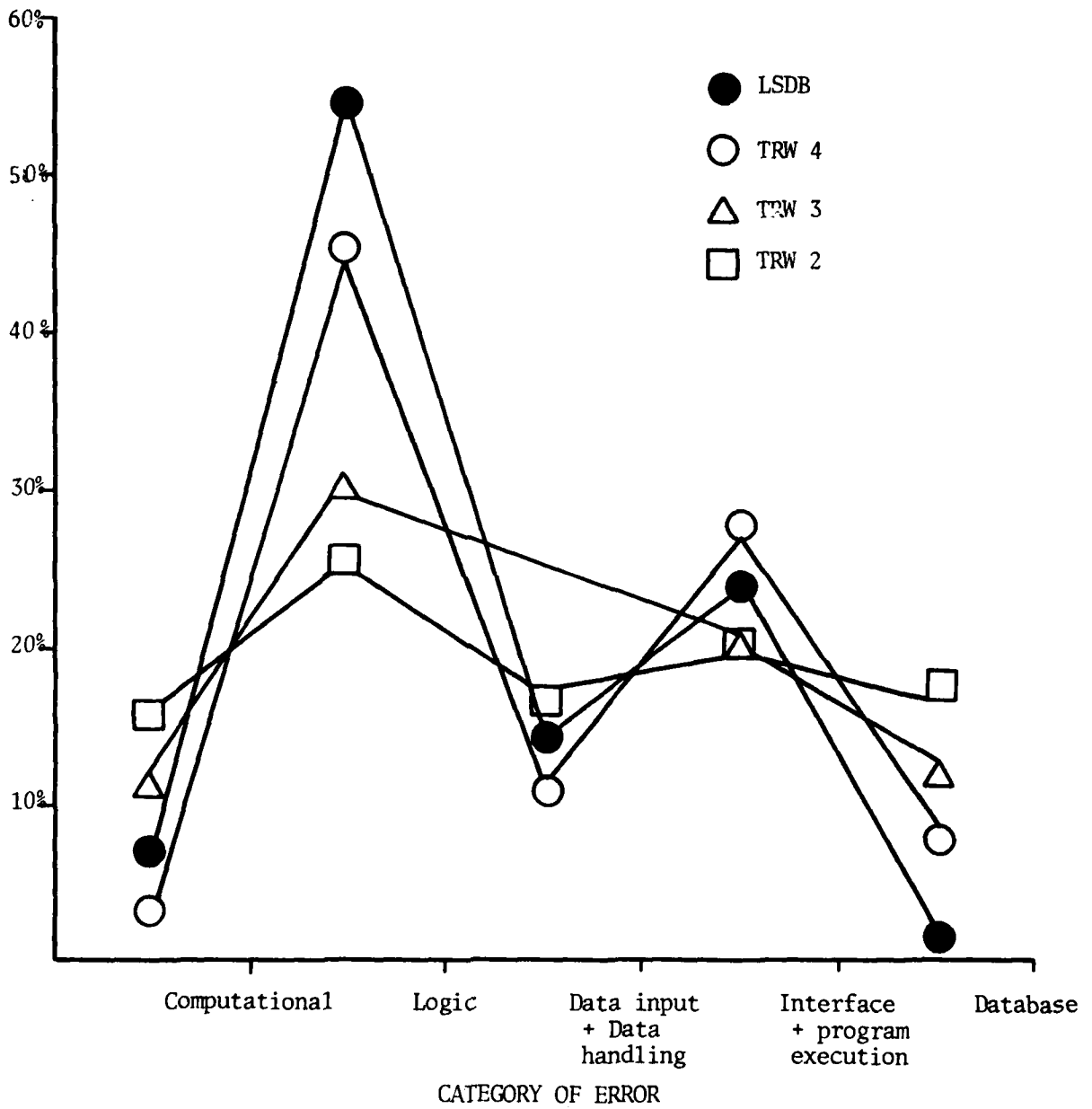


Figure 11. Comparison of error distributions between LSDB and three TRW studies

Table 9  
Frequencies of Runs and Errors by Subsystem

Subsystem	Number of Runs			% error ratio	
	Total	All Errors	Algorithmic errors	All	Algorithmic
LDI	299	45	7	15	2
LDG	419	90	33	21	8
LSO	458	97	27	21	6
BDP	510	112	18	22	4
BDR	315	57	16	18	5
A	533	71	29	13	5
B	185	36	10	19	5
Total	2719	508	140	19	5

integration work or development test runs, and are classified as categories A and B.

Relationships among the various categories in Table 9 were evaluated with the use of Kendall's tau ( $\tau$ ), a measure of correlation for rank-ordered data. This statistic was chosen to avoid the inflating effect extreme scores can often have on parametric statistics calculated on as few data points as represented by the number of subsystems. The rank orderings of total and executable lines were identical. However, the correlations of runs and error runs with lines of code were insignificant. Across the five subsystems and two unclassified categories, the number of runs correlated significantly with total errors ( $\tau = .71$ ) but not with algorithmic errors or error rates. The errors associated with each subsystem were unrelated to the Halstead metrics for volume, level, or effort computed on those subsystems (minimum scope case). Thus, it appears that while there was a relationship between the total number of runs and errors for a subsystem, these measures were unrelated to other characteristics of the subsystem such as size and complexity.

### 3.2.3 Error Trends over Time

A chronological history of the number of action reports, error-runs (total and algorithmic), and post-development errors is presented by month in Figure 12. The 63 action reports describing discrepancies between the design and the functional requirements were recorded only during the first seven months of development. Error-runs were reported over the entire 16 months of development. The distribution of total errors over months was bi-modal, with the first mode occurring during the second and third months of the project, and the second mode occurring between the ninth and eleventh months of the project. Post-development errors were first recorded during month 16 and continued until month 24. However, no activity was recorded from months 17 to 19.

The development error data indicated that work was performed on all subsystems nearly every month through the life of the project. These data suggest that the project team proceeded with the parallel development of subsystems. The alternative approach of depth first coding and implementation where one subsystem is completed before proceeding to the next, did not appear to have been employed.

Since the number of errors per month varied with the number of runs, a more representative measure of error generation was developed by dividing the number of errors by the number of runs. These rates for both total and algorithmic errors are plotted by month in Figure 13. Linear

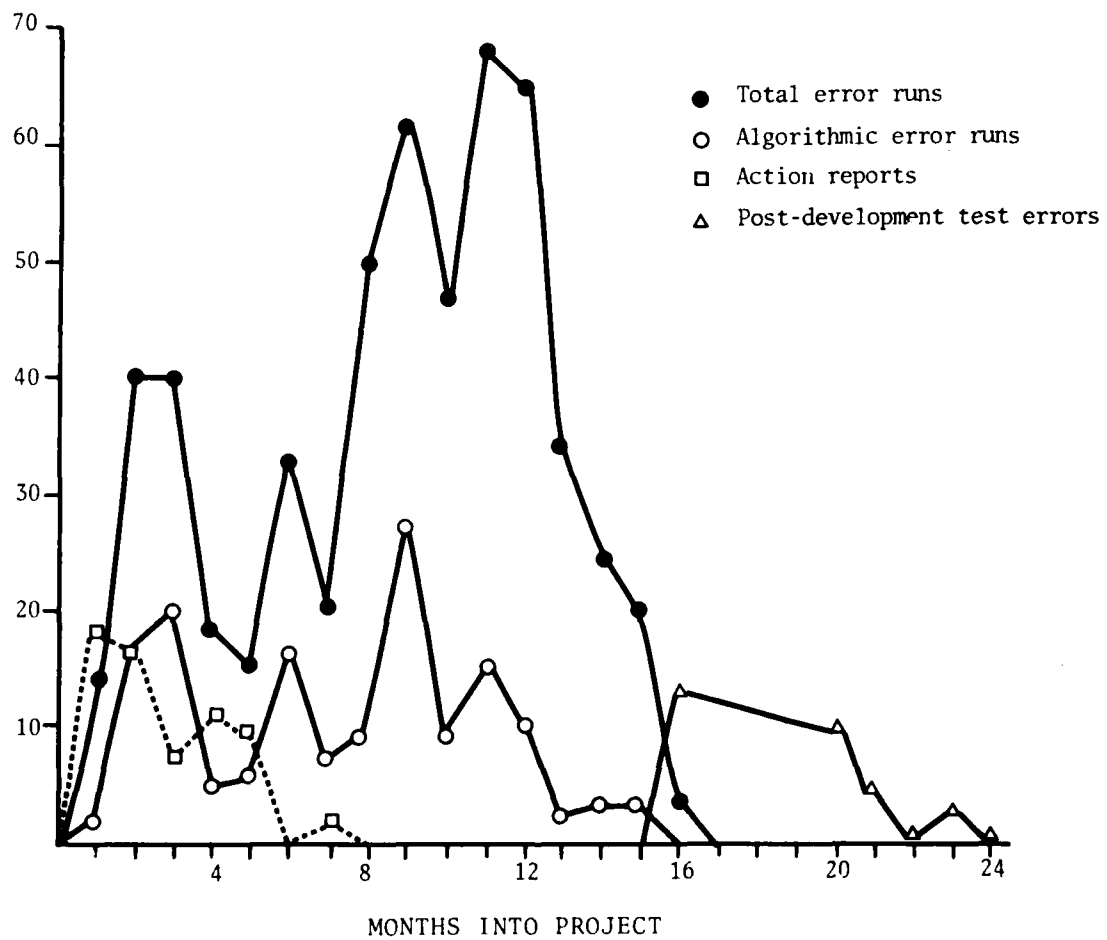


Figure 12. Action reports, error runs, and post-development test errors by month



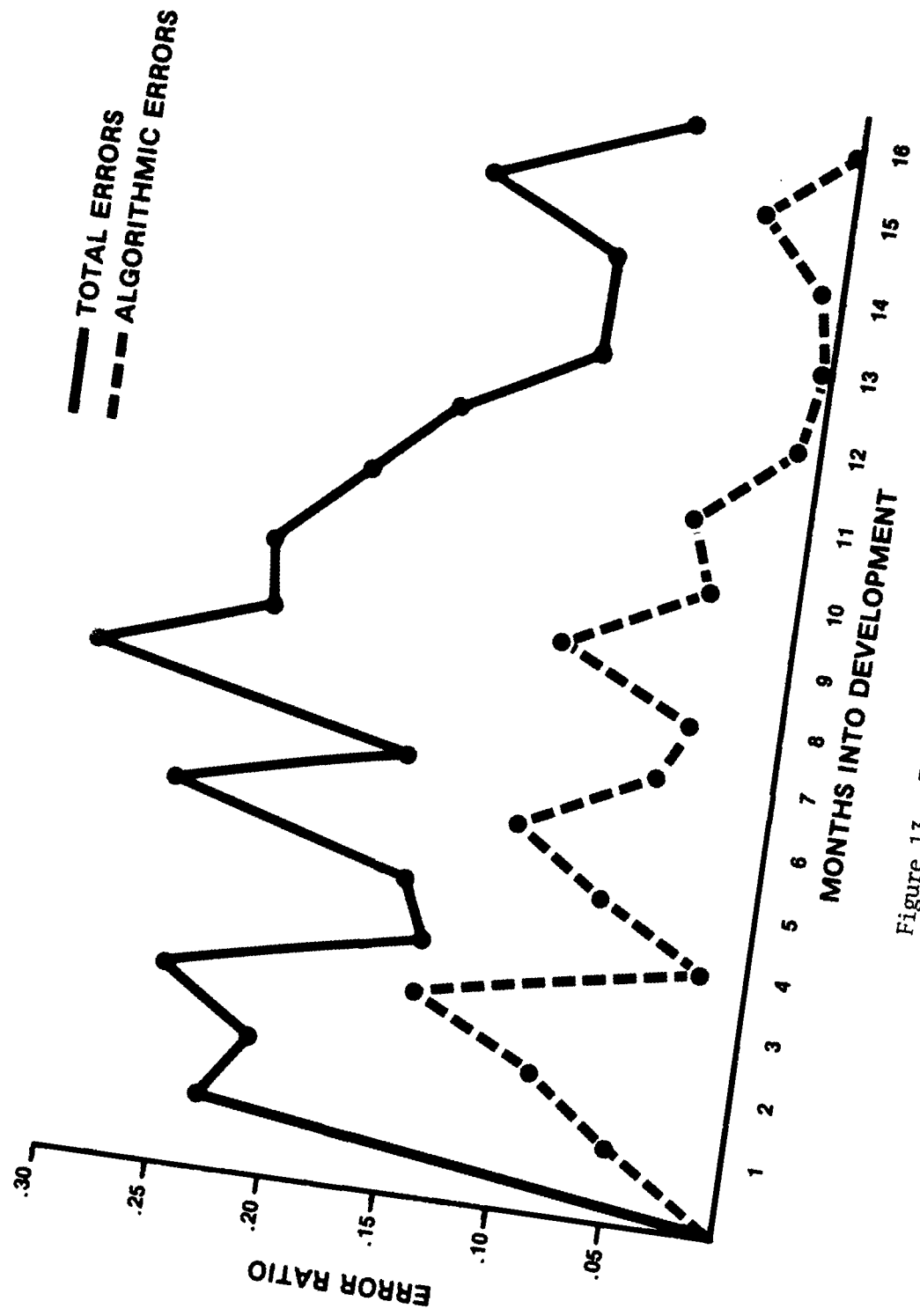


Figure 13. Error ratio by month

regressions for these error rates indicated decreasing trends over time. For total errors the correlation was  $-.52$ , while for algorithmic errors it was  $-.63$ . Rates for total errors decreased sharply over the final nine months from  $30\%$  in month 8 to  $8\%$  in month 16. The correlation associated with this sharp decline was  $-.90$  (predicted error rate =  $-2.5(\text{month \#}) + 47.78$ ).

### 3.2.1 Post-Development Errors

There were 28 post-development errors reported for the LSDB code, and as was evident in Figure 12, their frequency declined over time. These errors included 12 subsystem development test errors and 16 system integration test errors. Forty-three system integration test errors were reported for DAP. Compared to the size of the source code, proportionately fewer post-development errors were reported for LSDB. This comparison is even more striking because reports of subsystem development test errors were not available for DAP, thus the total number of post-development test errors for DAP should be even larger.

There are several methods of predicting the number of post-development errors from the kinds of data available here, the results of which are presented in Figure 14. Halstead's (1977) equations for the total number of delivered bugs led to a prediction of 27.2 errors for LSDB and 4 errors for DAP. Thus, the prediction was amazingly accurate for LSDB and substantially underestimated for DAP. Since much of Halstead's original work was performed on published algorithms, the accuracy of his predictions may improve with the quality of the code and the extent to which the code is an accurate reflection of the requirements and specifications. Such an explanation would be consistent with observations in these data, and with the fact that some requirements redefinition appears to have occurred during the DAP project.

The ratio of post-development errors to number of executable lines of code from each project was used to predict the number of post-development errors in the other project. The ratio for LSDB was  $0.0027$  and for DAP was  $0.0111$ . When the ratio for the LSDB project was multiplied by the number of executable lines in DAP, it led to a prediction of only 10 post-development errors for the DAP project (Figure 14). When the ratio for DAP was multiplied by the number of executable lines in LSDB, it led to a prediction of 116 post-development errors for LSDB. Thus, it is obvious that the proportion of post-development errors per line of executable code was four times greater for DAP than for LSDB.

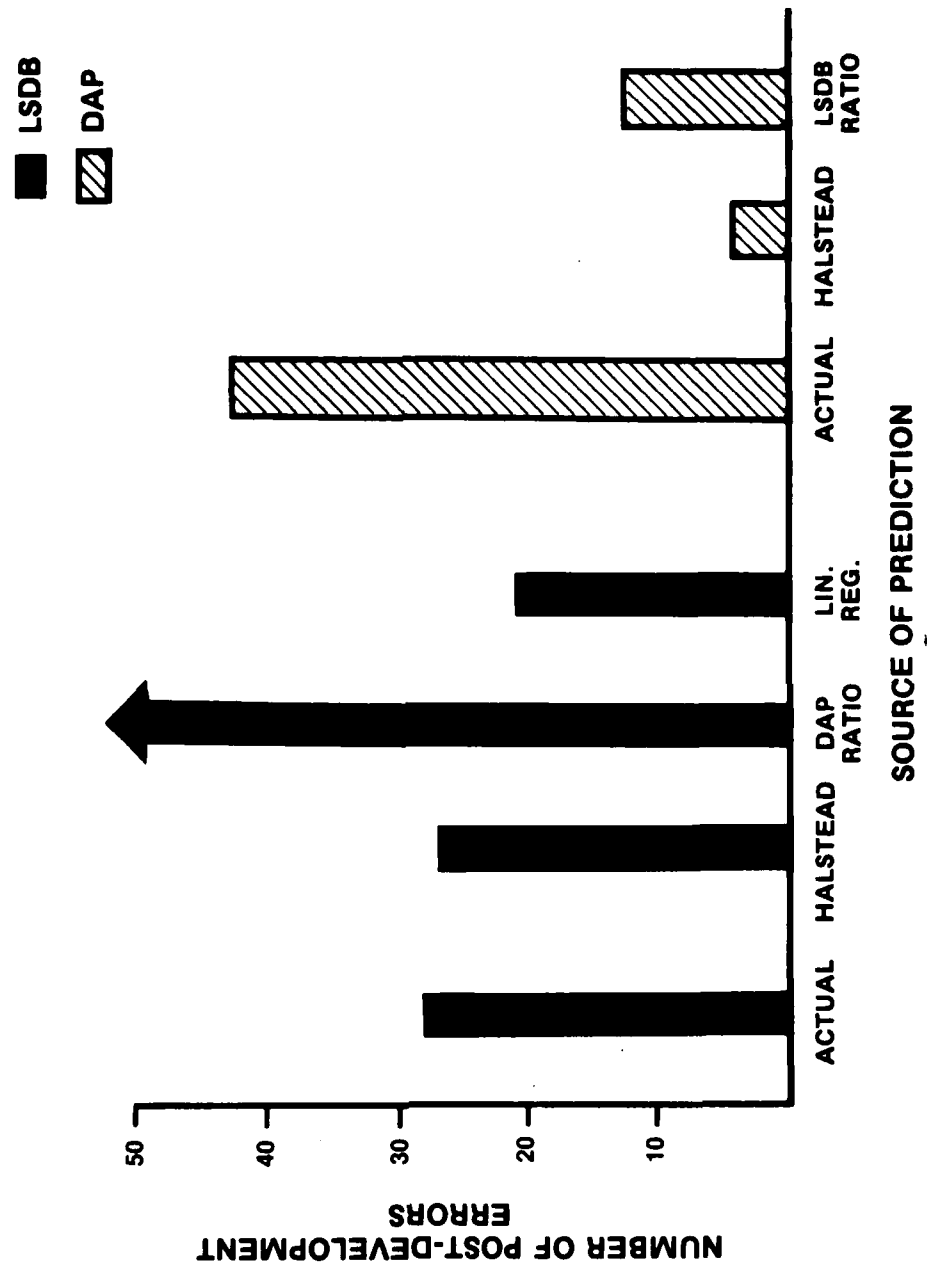


Figure 14. Prediction of post-development errors

An attempt was made to predict post-development errors from error-runs occurring in the development work performed prior to testing. As a simple heuristic in place of curve-fitting, a linear prediction equation was developed from the error rates occurring between months 8 and 15, since the standard error of estimate for error rates over this period was much smaller. Month 16 data was not included since post-development testing was initiated during this month. The six months during which post-development errors were reported were treated as continuous in this analysis since no activity appeared to have occurred during the three month gap between months 17 and 19 in the chronological data. The equation developed from the error rates recorded during months 8 through 15 was  $-2.52X + 48.02$ , where X represents the month number. In determining the predicted error rate for each month, this equation was applied only to months 16 through 19 in the chronological data. The regression line intercepts the X axis prior to month 20, predicting that no errors will be detected after month 19. When these error rates were multiplied by 170 runs (the average number of runs per month during the development phase), it was predicted that 26.5 errors would occur during months 16 to 19. The four development errors which occurred during month 16 were subtracted from this total, and the final prediction was that 22.5 post-development errors would be detected. This estimate is reasonably close to the 28 post-development errors actually reported.

#### 4. INTERVIEWS

The following general observations emerged from interviews with individuals associated with the LSDB project. On the basis of these interviews it appeared that the success enjoyed by this programming team was partly achieved by a fortuitous selection of team members. The particular personal styles of members were well-suited for their roles on the team and for the practices employed. The observations gleaned from these interviews are discussed below.

##### 4.1 Chief Programmer Teams

The team members interviewed felt that the most significant practice was the chief programmer team. In particular, they felt that the cohesiveness and the visibility of each member's work contributed to higher performance. It took several months of interaction before the team really became cohesive. The team felt that a strong source of morale resulted from keeping the nucleus of the team intact throughout the development cycle (contrary to ordinary practice at SAMTEC), with the possibility of continuing into future assignments. Since all team members shared in most tasks, no job (with the possible exception of librarian) was allowed to become too tedious or mundane.

Project participants felt the team needed its own physical space. The LSDB team had their own room with a lock on the door. The team worked around a conference table resulting in continuous interaction among team members, and ensured that the work of each member was consistent with the work of others. During afternoon breaks, the team would make popcorn, but some constraints were placed on their independence when an administrator found them making ice cream on a side porch.

The chief programmer was a software craftsman primarily interested in doing technical work without being hampered by administrative, personnel, or bureaucratic problems. He was an experienced software designer and coder who, because he understood the technical characteristics of the target system better than other project members, had the final say on system design and coding. Both the chief programmer and the tester were considered "range rats" (a local term for people who had considerable experience working on the Western Test Range at Vandenberg AFB). Over the years they had performed many jobs at Vandenberg and could anticipate many of the practical and technical problems they encountered during the software development. The chief programmer was a dynamic individual who was able to establish close working relationships with his customers.

The backup programmer had to work very closely with the chief programmer and their skills were supplementary. The backup programmer was quieter, a hard worker, and capable of generating large amounts of code. The backup did 80% of the coding for the system.

The librarian was responsible for taking the code and entering it into the system. The librarian was the only person who could actually change or add code to the system, although other members might have access to add data. The librarian was able to tolerate time constraints and maintain careful control throughout the project. The librarian was more than a keypuncher or "go-for", and was expected to develop some basic technical abilities such as setting up job control. The job level of the librarian was one step higher than an administrative specialist and several steps higher than clericals.

The procedure for submitting programs in this project required either the chief programmer or the backup to submit a written version of the program or change to the librarian who would enter it into the system, perform an initial check, eliminate any obvious errors, and then return a compiled listing of the program to the backup programmer. The backup or chief programmer would review this listing before performing the first execution. Careful records were kept of every submission, any resulting errors, and the code that had to be changed.

The tester worked along side the team almost as an alter ego. He set up the necessary data for both intermediate and final acceptance tests. It was important that the tester was not considered a member of the team. Nevertheless, the tester was not someone brought in at the last minute. From project initiation, he attended all the walk-throughs and became intimately familiar with the LSDB code.

Team members felt it was important that the team be able to select its members. Similarly, they felt the team should be able to oust members who were not compatible or who were not contributing to team efforts. The following list summarizes the team's recommendations for selecting members. The selection process should eliminate only incompetent programmers. A chief programming team represents a particular mix of skills and duties which may not be acquired if only brilliant programmers are selected (someone has to slog through the trenches). Some particular characteristics which seemed important to this team were:

### Chief Programmer

- dynamic
- excellent technical proficiency
- delegates red tape to someone higher up
- can establish close working relationships with customers
- has an almost paternal, protective (but democratic) attitude toward the team

### Backup Programmer

- less dynamic
- areas of technical competence supplement those of Chief Programmer.
- limitless capacity for generating code
- should be capable of filling in for Chief Programmer when necessary

### Support Programmer

- willing to participate in team interactions and work within team consensus
- cannot be a prima donna, a solitary worker, or unable to take criticism
- should be given a trial period before becoming a formal team member

### Librarian

- high tolerance for frustration and pressure
- willing to perform unending work
- needs some understanding of programming code
- needs typing skills

### Tester

- needs experience in content area of the program

- must be assertive
- should not become close knit with team

The team felt that new members would need some orientation to the working environment they were entering. Such training might be most effectively handled by experienced chief programmers rather than professional trainers. The type of people chosen for a team should be those who are adaptive to the types of new habits that will be the focus of training (this may be easier for less experienced programmers).

#### 4.2 Design and Coding Practices

LSDB was designed in a strictly top-down fashion and this practice was considered an important contribution to the success of the system. The chief programmer commented that in his 20 years of experience he had usually done top-down design, but that he had not employed top-down development. He considered them both to be important, especially when used in tandem. However, he reported problems in trying to implement the top-down development strategy within standard Air Force practices, especially the requirements for preliminary and critical design reviews. Considerable time was lost while waiting for the Configuration Control Board's approval of specifications.

In an attempt to implement modular design, most procedures in the system were constrained to a maximum of 100 lines of code or two pages of output including comments. There was an attempt to make each of the subroutines as functionally independent as possible and to restrict unnecessary data transmission between modules (G. Myers, 1978). The team favored the use of highly modular systems and believed this practice contributed significantly to the ease with which the resulting system could be maintained or modified. Although they thought structured coding was of benefit, they considered its relative importance to be small compared to that of modularity or the use of the chief programmer teams.

Variations in the construction of HIPO charts from guidelines described by IBM were identified by an independent validation and verification contractor and corrected. HIPO charts were very unpopular with both the team and the independent contractor. HIPO's were not maintained up-to-date throughout the project. The team felt that the HIPO charts were not particularly useful beyond the first or second level within the system hierarchy. HIPO's might have been more readily accepted had an interactive system been



available for generating them. The team recommended that a program design language (PDL) would have been much more useful than HIPO charts.

Walk-throughs were held every week and were attended by the software development team, the tester, the project administrator to whom the team reported, the customer, and the end users of the system. The team felt it was important that these walk-throughs were held weekly and that those involved in the procurement and use of the system were in attendance. In fact, since the team held code reviews internally on an ad hoc basis, they felt that walk-throughs were held primarily for the benefit of other interested parties.

Walk-throughs tended to last one to one and one-half hours. In most cases, the chief programmer handed out either the code or design description a week prior to its consideration so that all attendees could review the material and be prepared with detailed comments. They discovered early that walking through the code in detail was not practical. Rather, they gave high level descriptions of the routine's processing and went into the code only as required.

The team felt that the amount and type of documentation required was burdensome, especially the documentation of specifications and design. Both the developers and maintainers felt that the code was sufficiently well designed and documented internally that no other documentation was required. In no case had they gone back to the HIPO charts or even the specifications to obtain information in order to make a change to the system. Sophisticated documentation may not have seemed as important since most necessary modifications were so minor that a single statement could be isolated rather quickly.

As of March 1978 very little maintenance had been required. Since much of the maintenance has been minor, it has been suggested that the team librarian or someone with equivalent experience could make most of the one card modifications that have been required. The LSDB development team and tester received a letter of commendation from the end users. They were complimented both for the high quality of the software and for its production on time and within budget.

One anecdote is instructive on the effectiveness of the ASTROS guidelines. At the beginning of the project, the LSDB team was required to suspend work while waiting for the preliminary design review. In order to keep them from beginning to code, the SAMTEC engineer gave the team a

problem from another project. The program computed and plotted some range safety parameters. A tiger team had spent a month trying in vain to modify the existing program for an upcoming launch. The LSDB team redesigned the program and produced a working parameterized version in one week.

## 5. CONCLUSIONS

### 5.1 Current Results

The performance of the LSDB development project using the modern programming practices specified in the ASTROS plan was comparable to that of similar sized software development projects on numerous criteria. The amount of code produced per man-month was typical of conventional development efforts (however, this is a controversial measure of productivity; Jones, 1978). Nevertheless, the performance of the LSDB project was superior to that of a similar project conducted with conventional techniques in the same environment. Thus, the benefits of the modern programming practices employed on the LSDB project were limited by the constraints of environmental factors such as computer access and turnaround time.

While the results of this study demonstrated reduced programming effort and improved software quality for a project guided by modern programming practices, no causal interpretation can be reliably made. That is, with only two projects and no experimental controls, causal factors can only be suggested, not isolated. The ability to generalize these results to other projects is uncertain. For instance, it cannot be proven that modern programming practices had a greater influence on the results than differences among the individuals who comprised the LSDB and DAP project teams. Having acknowledged this restriction on causal interpretation, however, it is possible to weave together evidence suggesting that important benefits can be derived from the use of modern programming practices.

Several analyses demonstrated that improved efficiency was achieved through the use of the modern programming practices specified in the ASTRO plan. The value of Putnam's technology constant computed for LSDB was higher than for DAP. Further, the relative values of the LSDB and DAP projects on the parameters described in the RADC database consistently showed LSDB to have a higher performance than DAP when compared to projects of similar size, although the performance of both was close to the industry average. Since the DAP and LSDB projects shared similar processing environments, differences between the systems are probably not attributable to environmental factors.

The LSDB project demonstrated more efficient use of development man-hours than the DAP project. The Halstead parameters indicated that the LSDB project generated more code with less overall effort than DAP. LSDB exhibited a higher program level, indicating a more succinct representation of the underlying algorithm than was true for

DAP. The results of the effort analyses emphasize the great power in modular approaches to programming. If a programmer is required to keep the total context of a subsystem in mind, the time and effort required by the project increases. By breaking up the project into independent functional subroutines, the load upon the programmer is reduced.

Another area of evidence favoring modern programming practices was the superior quality of the LSDB code compared to that of DAP. Scores on the software quality metrics produced by McCall, Richards, and Walters verified that the practices employed on the LSDB project resulted in a more modularized design and structured code than DAP. Further, greater simplicity was evident in the LSDB code. Although the McCabe analysis indicated that the complexity and structuredness of the control flows were generally similar, the breaches of structured practice in the LSDB code were uniform. That is, the UNDO construct in S-FORTRAN allows branches out of conditions and loops. Although used consistently in the LSDB code, this construct is not among the structured programming practices recommended by Dijkstra (1972). This construct may not have made the control flow more difficult to understand (Sheppard, Curtis, Milliman, & Love, 1979). Departures from structured principles were far more varied in the DAP code, resulting in a convoluted control flow that is much more difficult to comprehend and trace.

Perhaps the most impressive comparison between LSDB and DAP concerns the number of post-development errors. When compared to DAP, the LSDB code contained three times as many lines and two and a half times as many executable lines, but only two-thirds as many post-development errors were reported for LSDB. The reliability achieved by LSDB was well predicted by both the Halstead equation for delivered bugs and a regression equation based on monthly error rates during development. The inability of any of the methods to predict post-development errors on DAP suggests that the prediction obtained for LSDB may have occurred by chance. Nevertheless, the existence of a tester associated with the LSDB development and the orientation of the project towards its final evaluation may have contributed to a strong correspondence between the requirements and the delivered code of LSDB. When this correspondence exists, the number of errors or error rate may prove much more predictable.

It is clear from analyses reported here that a software development project employing modern programming practices performed better and produced a higher quality product than a conventional project conducted in the same environment. However, these data do not allow the luxury of causal

interpretation. Even if such interpretation were possible, the data still do not allow analyses of the relative values of each separate practice. Further evaluative research will be required before confident testimonials can be given to the benefits of modern programming techniques. Nevertheless, the results of this study suggest that future evaluations will yield positive results if constraints in the development environment are properly controlled.

## 5.2 Future Approaches to Evaluative Research

This study demonstrated that exercising some control over the research environment can be extremely valuable. Without the comparable environment of LSDB and DAP this study would have found little evidence to indicate that modern programming practices have benefit. The experimental manipulation of selected practices (e.g., different ways of organizing programming teams) would improve future research efforts. Unless the separate effects of different practices can be identified, no recommendations can be made concerning them. Rather, the only conclusion that can be reached concerns the use of modern programming practices as a whole.

Data collection on programming projects often interferes with the programming task or meets opposition from team members. This problem can be counteracted by developing measurement tools embedded in the system which are invisible to programmers. Such tools would produce more reliable data since they cannot be forgotten, ignored, or incorrectly completed as manually completed forms frequently are. On the LSDB project, information such as number of runs and errors and time per run was acquired directly from the operating system. Manpower loadings can be taken from the hours recorded on time cards to be billed to the project, and can be broken down into job classifications. The program source code is also an excellent source of data concerning the quality and complexity of the code, and the number and types of statements. A program support library can keep track of the relative status of programs with minimal impact on programmers, the number of modules, and the time spent on each one.

Data collection can serve the purposes of both research and management. A software development manager needs visibility of project progress in order to control events and determine corrective action. A program support library can report module size, number of runs, and other summary information at regular intervals, and check project status to alert the manager to milestones. When these collection mechanisms are imposed unobtrusively on programming projects, their use is more likely to gain support, and better data will be available both to management and researchers.

## 6.0 REFERENCES

- Akiyama, F. An example of software system debugging. In Proceedings of the IFIPS Congress, 1971. Amsterdam: North Holland, 1971.
- Baker, F.T. Chief programmer team management of production programming. IBM Systems Journal, 1972, 11, 56-73.
- Baker, F.T., & Mills, H.D. Chief programmer teams. Datamation, 1973, 19 (12), 58-61.
- Barry, B.S., & Naughton, J.J. Structured Programming Series (Vol. 10) Chief Programmer Team Operations Description (RADC-TR-7400-300-Vol.X). Griffiss AFB, NY: Rome Air Development Center, 1975 (NTIS No. AD-A008 861).
- Belford, P.C., Donahoo, J.D., & Heard, W.J. An evaluation of the effectiveness of software engineering techniques. In Proceedings of COMPCON '77. New York: IEEE, 1977.
- Black, R.K.E. Effects of modern programming practices on software development costs. In Proceedings of COMPCON '77. New York: IEEE, 1977.
- Boehm, B.W. Software and its impact: A quantitative assessment. Datamation, 1973, 19 (5), 48-59.
- Brown, J.R. Modern programming practices in large scale software development. In Proceedings of COMPCON '77. New York: IEEE, 1977.
- Bulut, N., & Halstead, M.H. Impurities found in algorithm implementation. Sigplan Notices, 1974, 9 (3), 9-10.
- Caine, S.H., Farber, & Gordon, E.K. S-Fortran language reference guide. Pasadena, CA: Caine, Farber, & Gordon, Inc., 1974.
- Caine, S.H., Farber, and Gordon, E.K., S-Fortran programmers guide. Pasadena, CA: Caine, Farber, & Gordon, Inc., 1975.
- Cornell, L.M., & Halstead, M.H. Predicting the number of bugs expected in a program module (Tech. Rep. CSD-TR 205). West Lafayette, IN: Purdue University, Computer Science Department, October 1976.
- Curtis, B., Sheppard, S.B., & Milliman, P. Third time charm: Stronger prediction of programmer performance by software complexity metrics. In Proceedings of the 4th

International Conference on Software Engineering. New York: IEEE, 1979.

- Curtis, B., Sheppard, S.B., Milliman, P., Borst, M.A., & Love, T. Measuring the psychological complexity of software maintenance tasks with the Halstead and McCabe metrics. IEEE Transaction on Software Engineering, 1979, 5, 96-104.
- DeRoze, B.C. Software research and development technology in the Department of Defense. Paper presented at the AIIE Conference on Software, Washington, D.C., December 1977.
- DeRoze, B.C., & Nyman, T.H. The software life cycle - A management and technological challenge in the Department of Defense. IEEE Transactions on Software Engineering, 1978, 4, 309-318.
- Dijkstra, E.W. Notes on structured programming. In O.J. Dahl, E.W. Dijkstra, & C.A.R. Hoare (Eds.), Structured Programming. New York: Academic Press, 1972.
- Duvall, L.M. The design of a software analysis center. In Proceedings of COMPSAC '77. New York: IEEE, 1977.
- Elshoff, J.L. Measuring commercial PL/1 programs using Halstead's criteria. Sigplan Notices, 1976, 11, 38-46.
- Fitzsimmons, A.B., & Love, L.T. A review and evaluation of software science. ACM Computing Surveys, 1978, 10, 3-18.
- Funami, Y., & Halstead, M.H. A software physics analysis of Akiyama's debuqqing data. In Proceedings of the MRI 24th International Symposium: Software Engineering. New York: Polytechnic Press, 1976.
- Gordon, R.D., & Halstead, M.H. An experiment comparing Fortran programming times, with the software physics hypothesis. AFIPS Conference Proceedings, 1976, 45, 935-937.
- Gordon, R.D. A measure of mental effort related to program clarity. Unpublished doctoral dissertation, Purdue University, 1977.
- Halstead, M.H. Natural laws controlling algorithm structure. Sigplan Notices, 1972, 7 (2), 19-26.

- Halstead, M.H. Elements of Software Science. New York; Elsevier North-Holland, 1977.
- Hecht, H., Sturm, W.A., & Trattner, S. Reliability measurement during software development (NASA-CR-145205). Hampton, VA: NASA Langley Research Center, 1977.
- Jones, T.C. Measuring programming quality and productivity. IBM Systems Journal, 1978, 17 (1), 39-63.
- Katzen, H. Systems Design and Documentation: An Introduction to the HIPO Method. New York: Van Nostrand Reinhold, 1976.
- Lyons, E.A., & Hall, R.R. ASTROS: Advanced Systematic Techniques for Reliable Operational Software. Lompoc, CA: Vandenburg, AFB, Space and Missile Test Center, 1976.
- Maxwell, F.D. The determination of measures of software reliability (NASA-CR-158960). Hampton, VA: NASA Langley Research Center, 1978.
- McCabe, T.J. A complexity measure. IEEE Transactions on Software Engineering, 1976, 2, 308-320.
- McCall, J.A., Richards, P.K., & Walters, G.F. Factors in software quality (Tech. Rep. 77C1S02). Sunnyvale, CA: General Electric, Command and Information Systems, 1977.
- Mills, H.D. Mathematical foundations for structured programming. In V.R. Basili and T. Baker (Eds.), Structured Programming. New York: IEEE, 1975.
- Myers, G.J. Composite/Structured Design. New York: Van Nostrand Reinhold, 1978.
- Myers, W. The need for software engineering. Computer, 1978, 11 (2), 12-26.
- Nelson, R. Software data collection and analysis (Unpublished report). Rome, NY: Griffiss AFB, Rome Air Development Center, 1978.
- Norden, P.V. Project life cycle modeling: Background and application of the life cycle curves. In Software Phenomenology: Working Papers of the Software Life Cycle Management Workshop. Atlanta: U.S. Army Institute for Research in Management Information and Computer Science, 1977.



- Ottenstein, K.J. A program to count operators and operands for ANSI-Fortran modules (Tech. Rep. 196). West Lafayette, IN: Purdue University, Computer Science Department, 1976.
- Putnam, L.H. A general empirical solution to the macro software sizing and estimating problem. IEEE Transactions on Software Engineering, 1978, 4, 345-361.
- Salazar, J.A., & Hall, R.R. ASTROS Advanced Systematic Techniques for Reliable Operational Software: Another Look. Lompoc, CA: Vandenburg, AFB, Space and Missile Test Center, 1977.
- Shannon, C.E. A mathematical theory of communication. Bell System Technical Journal, 1948, 27, 379-423.
- Sheppard, S.B., Curtis, B., Milliman, P., & Love, T. Human factors experiments on modern coding practices. Computer, 1979, 12, in press.
- Stay, J.F. HIPO and integrated program design. IBM Systems Journal, 1976, 15, 143-154.
- Stevens, W.P., Myers, G.J., & Constantine, L.L. Structured design. IBM Systems Journal, 1974, 13, 115-139.
- Stroud, J.M. The fine structure of psychological time. Annals of the New York Academy of Sciences, 1966, 623-631.
- Tausworthe, R.C. Standardized Development of Computer Software (2 vols.). Englewood Cliffs, NJ: Prentice-Hall, 1979.
- Thayer, T.A., et al. Software reliability study (RADC-TR-76-238). Rome, NY: Griffiss AFB, Rome Air Development Center, 1976, A030798.
- Walston, C.E., & Felix, C.P. A method of programming measurement and estimation. IBM Systems Journal, 1977, 18 (1), 54-73.
- Yourdon, E., & Constantine, L.L. Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design. Englewood Cliffs, NJ: Prentice-Hall, 1979.
- Yourdon Report (Vol. 1, No. 9). New York: Yourdon, Inc. 1976.

APPENDIX A

DATA COLLECTION FORMS

SYSTEM LSDB DATE 1/21/76  
GENERAL CONTRACT/PROJECT SUMMARY

1. Type of Contract: FFP \_\_\_\_\_ CPFF \_\_\_\_\_ OTHER X \_\_\_\_\_
2. Total Cost (Actual or Estimated) \_\_\_\_\_
3. Level of Subcontracting NONE
4. Project Environment
  - Dev. Team Collocated with User? NO
  - Dev. Team Collocated with Computers? NO
  - Dev. System Same as Operational Systems? YES
  - Test & Integration Separate Organization? YES
5. Project Description: LSDB prepares and updates all parameters needed by the Real Time Segment (RTS) of Project MIPS and the Backup Information Display System (BIDS)
6. Est. Start Date 10/15/75 Est. End Date July 1977
7. Est. Number of Project Personnel

Management	<u>1</u>	Design & Analysis	<u>1</u>
Support	<u>1</u>	Programming est.	<u>5 actual 2</u>
		Test	<u>1</u>
8. Est. Number of CPC's 5
9. Est. Number Pages of Documentation:

Requirements	<u>100</u>	Test Doc's	<u>50</u>
Specifications	<u>100</u>	User Manuals	<u>50</u>
10. Est. Total Number of Instructions 4000 Fortran 1000 Bal
11. Est. Number of Different Input Formats 19
12. Est. Number of Different Output Formats 6
13. Est. Total Number of Man/Months:

Management	<u>16</u>	Design & Analysis	<u>8</u>
Support	<u>16</u>	Programming	<u>64</u>
Test	<u>8</u>		
14. Est. Total Computer Time (HRS) 360 hours

Contact \_\_\_\_\_

GENERAL CONTRACT/PROJECT SUMMARY  
INSTRUCTIONS

To be filled out at contract award or work request receipt by the chief programmer or project lead. This form will provide a general feel for the relative size and complexity of the project.

1. In a non-contract environment, put work request number under other.
2. Actual or estimated (identify which) total budget for the complete project.
3. Identify if more than one vendor is involved in development.
4. Yes or No response. User collocation means same facility. Computer collocation means same building.
5. Brief abstract and/or document references. Include such information as whether project is a conversion or new development, whether its real time or non-real time, and any unusual constraints placed upon development (hardware delivery, special schedules, unique data entry procedures, etc.). Attach extra sheet, if needed.
6. Project start date and est. completion date (date system is scheduled to become operational) in the form day-month-year.
7. Self-explanatory.
8. Estimate of computer program components to be developed.
9. Includes all documentation (Design & Develop. Specs, User Manuals, Test Plan, etc.).
10. Both for higher level languages and assembler code.
11. Includes card, tape, disk data calls.
12. Includes outputs to all peripheral devices.
13. Includes design, program, test as well as management and support.
14. Includes pre-compile, compile, assembly, debug, test & integration.

SYSTEM LSDB

DATE 21 January 76

MANAGEMENT METHODOLOGY SUMMARY

1. Management Procedures/Tool Used \_\_\_\_\_
  
2. List Reports Generated Including all Specs and Requirement Documents.
  - a. Development Spec      Supplied to Customer      Yes
  - b. Product Spec              Supplied to Customer      Yes
  - c. Test Plan                      Supplied to Customer      Yes
  - d. Test Procedures              Supplied to Customer      Yes
  - e. Final Report                  Supplied to Customer      Yes
  
3. Formal Reviews and Schedule
  - a. PDR (Structured Walk Thru)      Date      June 1976
  - b. CDR (Structured Walk Thru)      Date      Sept 1976
  - c. \_\_\_\_\_                      Date      \_\_\_\_\_
  - d. \_\_\_\_\_                      Date      \_\_\_\_\_
  
4. AF Regulations, Manuals, and Military Standards Under Which Development Will Be Conducted.  
\_\_\_\_\_
  
5. Description of Deliverable Software      LSDB CPCI  
\_\_\_\_\_
  
6. Reference Measurement Gathering Procedures \_\_\_\_\_  
\_\_\_\_\_

MANAGEMENT METHODOLOGY SUMMARY  
INSTRUCTIONS

To be filled out by chief programmer or project lead at contract award or upon work request receipt. This form will establish review schedule and guideline documentation.

1. List any procedures and/or automated tools which will be utilized to increase management control and visibility.
2. List all technical documentation available at CDR which will serve as a guideline for development. Note if these documents are to be provided to the customer.
3. List formal reviews and tentative schedules.
4. Reference all government documents to be used as guideline to development, testing, review and documentation.
5. Briefly describe the deliverable software package including any special tools, assemblers, executives etc., which will be supplied to customer.
6. Reference document(s) which provide measurement gathering procedures/responsibilities.

SYSTEM LSDB

DATE 28 June 76

DESIGN AND PROCESSOR SUMMARY

1. Target Computer(s) 360-65  
Target Computer Same as Development Computer Yes
2. Processing Environment \_\_\_\_\_
3. Configuration: On Line Batch X Remote Batch X
4. Operating System(s) Version 21.7 IBM OS
5. Compiler Version(s) H FORTRAN
6. Assembler(s) F
7. Est. Percent: Hol 90 % Assembler 10 %
8. Automated Software Tools Used  
S-FORTRAN  
LIBRARIAN (Attach Vendors User Manuals)
9. Design Standards \_\_\_\_\_
10. Programming Standards ASTROS PLAN
11. Programming Techniques Employed:  
Top Down Design X HIPO \_\_\_\_\_ X  
Chief Programmer X Structured Code \_\_\_\_\_ X  
Librarian \_\_\_\_\_ X Structured Walk Thru \_\_\_\_\_ X  
Top Down Test \_\_\_\_\_ X Other Program Support  
Library
12. List Existing Programs/CPC's to be Used Subroutine  
Library
13. Est. Turnaround Time (HRS): Batch 24 Remote Batch 2  
Contact \_\_\_\_\_

DESIGN AND PROCESSOR SUMMARY  
INSTRUCTION

To be filled out by chief programmer or project lead when project is ready for Critical Design Review (Before coding starts). This form will define the processing environment and the technologies to be employed.

1. List all computers to be used for development, test and integration by manufacturers model number. Note if target and development computers are the same.
2. List all Peripherals to be employed for I/O processing and temporary storage. Note any Unique Data Entry/Display Techniques to be used. Note any special interpretive simulators, read-only memories, firmware, etc., to be used or built for this project.
3. Check all job submittal methods to be employed.
4. List all operating systems and versions.
5. List compilers and levels for all higher level languages employed, if applicable.
6. List all assembly languages to be used.
7. Estimate the percentage of object code to be generated by both HOL's and assembly languages.
8. List any special automated tools to be employed for assisting design coding, auditing, testing, measuring, documenting, and management control.
9. Reference Design Standards Manual(s).
10. Reference Programming Standards Manual(s).
11. Check techniques to be employed.
12. Reference all guidelines, implementation plans, program management plans, procedures, etc., which will be used for management controls and procedures.
13. Estimate average turnaround time for job submittal.

Definition: Turnaround time is the total working hours from the time the job leaves the developers hands until the time it is returned to him, minus any system down time.



SYSTEM \_\_\_\_\_

DATE \_\_\_\_\_

TESTING SUMMARY

1. Requirements and Specification Documents \_\_\_\_\_  
\_\_\_\_\_
  2. Test Plans/Procedures \_\_\_\_\_  
\_\_\_\_\_
  3. Testing Philosophy \_\_\_\_\_  
\_\_\_\_\_
  4. Method Employed to Audit Coding Standard Adherence \_\_\_\_\_  
\_\_\_\_\_
  5. Formal Audits and Dates
    - a. \_\_\_\_\_ Date \_\_\_\_\_
    - b. \_\_\_\_\_ Date \_\_\_\_\_
    - c. \_\_\_\_\_ Date \_\_\_\_\_
  6. Internal Management Procedures for Control of Testing \_\_\_\_\_  
\_\_\_\_\_
  7. Quality Assurance Procedures \_\_\_\_\_  
\_\_\_\_\_
- Contact \_\_\_\_\_

## TESTING SUMMARY INSTRUCTIONS

To be filled out by chief programmer or member of independent test group when project is ready for system test. This form identifies testing and requirements documents. It also identifies testing approaches, tools, procedures, etc.

1. Reference all requirements and specification documents which were the guidelines for system development.
2. Reference all test plans/procedures the system is being tested against.
3. Briefly describe the overall testing philosophy including debug, computer program test, and integration.
4. Briefly describe or reference procedure documentation for verifying coding standard adherence.
5. List formal audits and tentative dates.
6. Reference management procedures used as guideline for testing.
7. Reference quality assurance documents which describe QA involvement in testing.
8. Estimate man-hours to be expended in testing, include programmers, testers, QA, support and management.

SYSTEM \_\_\_\_\_

DATE \_\_\_\_\_

GENERAL PROJECT WRAP-UP REPORT

1. Total Computer Resources Used (HRS) \_\_\_\_\_
2. Man-Hours Charged to Projects:  
Management \_\_\_\_\_ Programming \_\_\_\_\_  
Design & Analysis \_\_\_\_\_ Support \_\_\_\_\_  
Test \_\_\_\_\_
3. Project Completion Date \_\_\_\_\_
4. Actual Number of CPC's \_\_\_\_\_
5. Total Lines of Code: HOL \_\_\_\_\_ Assembly \_\_\_\_\_
6. Total Size of Object Program (Specify Units) \_\_\_\_\_
7. Number of Pages Documentation \_\_\_\_\_
8. Number of Different Input Formats \_\_\_\_\_
9. Number of Different Output Formats \_\_\_\_\_
10. Actual Turnaround Time: Batch \_\_\_\_\_ Remote Batch \_\_\_\_\_
11. Total Errors in Each Error Category:  
A. Computational Error \_\_\_ G. Array Processing Errors \_\_\_  
B. Logic Errors \_\_\_\_\_ H. Data Base Errors \_\_\_\_\_  
C. Data Input Errors \_\_\_\_\_ I. Operation Errors \_\_\_\_\_  
D. Data Handling Errors \_\_\_ J. Program Execution Errors \_\_\_  
E. Data Output Errors \_\_\_\_\_ K. Documentation Errors \_\_\_\_\_  
F. Interface Errors \_\_\_\_\_ L. Other \_\_\_\_\_
12. Turnover Rate:  
Chief Programmer \_\_\_\_\_  
Back-Up Programmer \_\_\_\_\_  
Librarian \_\_\_\_\_  
Other Project Personnel \_\_\_\_\_

Contact \_\_\_\_\_

GENERAL PROJECT WRAP-UP REPORT  
INSTRUCTIONS

To be filled out by the chief programmer or project lead after the system has been bought by the user. This form will provide "ACTUALS" to all the estimates made on forms completed earlier in the development cycle.

1. Give the total computer time for development, test, and integration to the nearest tenth of an hour.
2. Provide total man-hours expended in each of the four categories, "SUPPORT" should include librarian, clerical, technical editing, QA, etc.
3. Date system became operational in the form day-month-year.
4. Number of computer program components in system.
5. Do not count comments or repeated common and data declaration statements.
6. If figure given in words, specify number of bits per word.
7. This should include all delivered documentation generated in support of this project.
8. Includes, card, tape, disk data calls.
9. Includes outputs to all peripheral devices.
10. Turnaround time for job submittal.

Definition: Turnaround time is the total working hours from the time the job leaves the developers hands until the time it is returned to him, minus any system down time.

11. These are the sums from all the COMPUTER PROGRAM FAILURE ANALYSIS REPORT forms.
12. Turnover rate: Number of man-months each person held the position. Use separate sheet if necessary.

SYSTEM \_\_\_\_\_

DATE \_\_\_\_\_

PERSONNEL PROFILE

1. Name or Id \_\_\_\_\_
2. Project Assignment (Job Title) \_\_\_\_\_
3. Education Level: HS \_\_\_\_ YRS \_\_\_\_ College \_\_\_\_ YRS  
Degree(s) \_\_\_\_\_
4. Special Computer Training Courses:
  - a. \_\_\_\_\_ Date \_\_\_\_\_
  - b. \_\_\_\_\_ Date \_\_\_\_\_
  - c. \_\_\_\_\_ Date \_\_\_\_\_
5. Target Language(s) \_\_\_\_\_
6. Years of Experience as:  
Operator/Technical \_\_\_\_\_ Analyst \_\_\_\_\_  
Programmer \_\_\_\_\_ Other \_\_\_\_\_
7. Years of Experience on:  
Target Computer(s) \_\_\_\_\_ Target Language \_\_\_\_\_  
Operating System \_\_\_\_\_ Similar Projects \_\_\_\_\_
8. List Other Programming Languages \_\_\_\_\_  
\_\_\_\_\_
9. List Other Computers \_\_\_\_\_  
\_\_\_\_\_

## PERSONNEL PROFILE INSTRUCTIONS

To be filled out by all project personnel when they are assigned to the project. This form will provide the education and experience levels of the personnel assigned to the project.

1. Names are not necessary. We're evaluating the technology not the people.
2. Persons assignment description or job title, if meaningful.
3. List years completed and degrees attained, with majors.
4. Title of specialized computer or programming training courses, with approximate date completed (month and year).
5. The total of these four categories should equal total years technical experience.
6. "Similar Projects" means projects requiring same amount of mathematical knowledge, using similar I/O processors, having equivalent data base requirements, of same relative complexity, etc.
7. List other programming languages with which you have coded.
8. List other computers you have written programs for, by vendor and model number.

SYSTEM \_\_\_\_\_

DATE \_\_\_\_\_

TECHNOLOGY CRITIQUE

1. Name or ID \_\_\_\_\_
2. Project Assignment (Job Title) \_\_\_\_\_
3. Time on Project \_\_\_\_\_

4. Techniques/Tools Used:

Top Down Design \_\_\_\_\_  
Top Down Test \_\_\_\_\_  
Structured Code \_\_\_\_\_  
HIPO \_\_\_\_\_  
Program Design Language \_\_\_\_\_  
Structured Walk Thru \_\_\_\_\_  
Other \_\_\_\_\_  
\_\_\_\_\_  
Preprocessor \_\_\_\_\_  
Program Support Library \_\_\_\_\_  
Other \_\_\_\_\_

5. List techniques/tools which seemed to yield productivity/reliability benefits, and you would like to use again. (Attach separate sheet if more room needed).  
\_\_\_\_\_  
\_\_\_\_\_
6. Recommend changes to organization/procedures/technology application which should be made before attempting another project. (Attach separate sheet if more room needed).  
\_\_\_\_\_  
\_\_\_\_\_
7. List special training received for this project and comment on its adequacy. (Attach separate sheet if more room is needed).  
\_\_\_\_\_  
\_\_\_\_\_

### TECHNOLOGY CRITIQUE INSTRUCTIONS

To be filled out by all project personnel as they leave the project or when the project is complete. This form will provide a subjective evaluation of the technologies employed by the people who actually used them.

1. Names are optional.
2. Person's assignment description or job title, if meaningful.
3. List time assigned to project in months.
4. Check techniques/tools used on project, briefly describe others.
5. Honest evaluation of your feelings.
6. Be candid, help us define a workable policy.
7. Once again be candid, lets make the training useful.



SYSTEM \_\_\_\_\_

DATE \_\_\_\_\_

SYSTEMS DEVELOPMENT LOG

1. Event \_\_\_\_\_
2. Phase in Life Cycle and Date \_\_\_\_\_
3. Major Decisions/Results \_\_\_\_\_  
\_\_\_\_\_
4. Factors Determining the Decision/Results \_\_\_\_\_  
\_\_\_\_\_
5. Delivered Product(s) \_\_\_\_\_  
\_\_\_\_\_
6. Other Documentation Related to this Event:
  - a. \_\_\_\_\_ Supplied to Customer \_\_\_\_\_
  - b. \_\_\_\_\_ Supplied to Customer \_\_\_\_\_
  - c. \_\_\_\_\_ Supplied to Customer \_\_\_\_\_
7. Personnel Involved (Name/Organization) \_\_\_\_\_  
\_\_\_\_\_  
\_\_\_\_\_
8. Est. Schedule Impact \_\_\_\_\_

Contact \_\_\_\_\_

## SYSTEMS DEVELOPMENT LOG INSTRUCTIONS

To be filled out by chief programmer, backup programmer or responsible developer whenever a significant event occurs. These events include documentation delivery, reviews, audits, turning over a CPC for integration testing, formal demonstration, or any design decision which may impact developmental schedule. This multi-purpose form will chart the project progress and provide a historical record of all significant events.

1. Describe the event, such as, delivery of a document, critical design review, redesign of CPC XYZ, etc.
2. Phase refers to the traditional systems management phases of software life cycle, i.e., concept, validation, development and operational.
3. Describe planned course of action or result of action, whichever is applicable.
4. Describe cause of the event. (This could simply be a schedule commitment).
5. Reference document(s) delivered as result of event, if applicable.
6. Reference support documentation which either precipitated the event or provides background for event.
7. List all personnel, both customer and developer, involved in event.
8. If applicable, estimate positive or negative effect event will have on schedule.

SYSTEM \_\_\_\_\_

DATE \_\_\_\_\_

COMPUTER PROGRAM RUN ANALYSIS REPORT

1. Computer Program Component ID \_\_\_\_\_

2. Run Date: \_\_\_\_\_ Day \_\_\_\_\_ Mon \_\_\_\_\_ Yr \_\_\_\_\_ Hr \_\_\_\_\_ Min

3. Successful Run? \_\_\_\_\_

4. CPU Time: \_\_\_\_\_ Min \_\_\_\_\_ Sec

5. Category of Work:

- a. Program Development \_\_\_\_\_
- b. Program Modification: \_\_\_\_\_
  - (1) Implementation of Additional Requirement \_\_\_\_\_
  - (2) Implementation of Hardware Change \_\_\_\_\_
  - (3) Memory/Time Optimization Enhancement \_\_\_\_\_
  - (4) Error Correction \_\_\_\_\_
  - (5) Design Modification \_\_\_\_\_
- c. Program Conversion \_\_\_\_\_
- d. Other \_\_\_\_\_

6. CPCI/CPC Status

- a. CPC Test and Eval \_\_\_\_\_
- b. Partial Integ. Test \_\_\_\_\_
- c. Full Integ. Test \_\_\_\_\_
- d. Production Program \_\_\_\_\_
- e. Other \_\_\_\_\_

7. Program Activity

- a. Compilation \_\_\_\_\_
- b. Compile and run \_\_\_\_\_
- c. Run with no compile \_\_\_\_\_
- d. Other \_\_\_\_\_

8. Number of Source Statements Changed/Deleted Inserted

- a. None \_\_\_\_\_
- b. 1-10 \_\_\_\_\_
- c. 11-20 \_\_\_\_\_
- d. 21-30 \_\_\_\_\_
- e. 31-40 \_\_\_\_\_
- f. 41-50 \_\_\_\_\_
- g. 51-75 \_\_\_\_\_
- h. 76-100 \_\_\_\_\_
- i. 101-150 \_\_\_\_\_
- j. 151-200 \_\_\_\_\_
- k. Over 200 \_\_\_\_\_

Contact \_\_\_\_\_

COMPUTER PROGRAM RUN ANALYSIS REPORT  
INSTRUCTIONS

To be filled out by programming librarian or responsible programmer after each computer run. If the run was unsuccessful (SYNTAX errors, abort, calculation error, loop, etc.), the supplemental form COMPUTER PROGRAM FAILURE ANALYSIS REPORT should also be complete. This form will yield error statistic data and computer run time data.

1. Use program mnemonic.
2. This time is start time of computer execution from the computer printout.
3. If answer is no, complete COMPUTER PROGRAM FAILURE ANALYSIS REPORT.
4. This can be gotten from the computer printout.
5. Check the appropriate box.
6. Check the appropriate box.
7. Check the appropriate box.
8. Check the appropriate box.

SYSTEM \_\_\_\_\_

DATE \_\_\_\_\_

COMPUTER PROGRAM FAILURE ANALYSIS REPORT

1. Computer Program Component ID \_\_\_\_\_

2. Run Date: \_\_\_\_ Day \_\_\_\_ Mon \_\_\_\_ Yr \_\_\_\_ Hr \_\_\_\_ Min

3. Severity of Failure

- A. Caused Complete System to Crash \_\_\_\_\_
- B. Caused A Dependent Job to Fail \_\_\_\_\_
- C. Local Job Failure Only \_\_\_\_\_
- D. Real Time Failure \_\_\_\_\_
- E. Other \_\_\_\_\_

4. Error Category Count

- A. Computational Error \_\_\_\_\_
- B. Logic Error \_\_\_\_\_
- C. Data Input Error \_\_\_\_\_
- D. Data Handling Error \_\_\_\_\_
- E. Data Output Error \_\_\_\_\_
- F. Interface Error \_\_\_\_\_
- G. Array Processing Error \_\_\_\_\_
- H. Data Base Error \_\_\_\_\_
- J. Program Execution Error \_\_\_\_\_
- K. Documentation Error \_\_\_\_\_
- L. Other \_\_\_\_\_

Contact \_\_\_\_\_

COMPUTER PROGRAM FAILURE ANALYSIS REPORT  
INSTRUCTIONS

To be filled out by the responsible developer for each unsuccessful run. The failure information should be available on the program printout or from the computer operator. The error data can be derived from an analysis of the program output. (It is possible that a failure can be caused by more than one error, list them all).

1. Use program mnemonic.
2. This time is start time of computer execution from the computer printout.
3. Check box which most nearly describes the failure indication. If other is checked, briefly describe failure.
4. The count under the error category means number of errors not number of erroneous statements.
  - A. Examples of COMPUTATIONAL ERRORS include: (1) Incorrect operand in equation, (2) Incorrect use of parenthesis, (3) Sign convention error (4) Units or data conversion error, (5) Computation produces an over/under flow, (6) Incorrect equation used, (7) Precision lost due to mixed mode, (8) Missing computations, (9) Rounding or truncation error and loop.
  - B. Examples of LOGIC ERROR include: (1) Incorrect operand in logical expression (2) Logic activities out of sequence, (3) Wrong variable being checked, (4) Missing logic or condition tests, (5) Too many/too few statements in loop, (6) Loop iterated incorrect number of times (including endless loop).
  - C. Examples of DATA INPUT ERRORS include: (1) Invalid input read from correct data file, (2) Input read from incorrect data file, (3) Incorrect input format, (4) Incorrect format statement referenced, (5) EOF encountered prematurely, (6) EOF missing.
  - D. Examples of DATA HANDLING ERRORS include: (1) Data file not rewound before reading, (2) Data initialization not done, (3) Data initialization done improperly, (4) Variable used as a flag or index not

set properly, (5) Variable referred to by wrong name, (6) Variable type is incorrect, (7) Data packing/unpacking error, (8) Sort error, (9) Subscripting error.

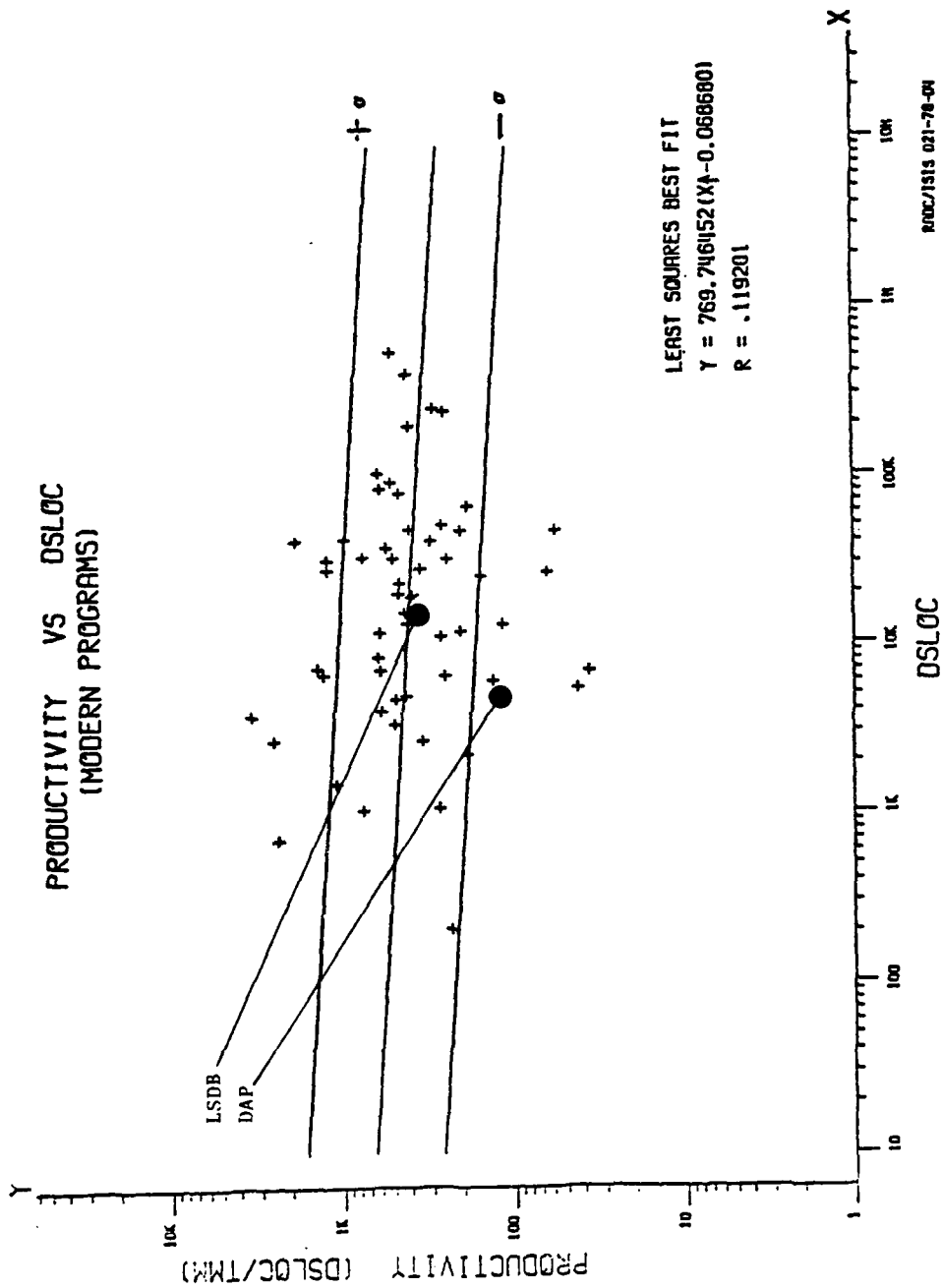
- E. Examples of DATA OUTPUT ERRORS include: (1) Data written on wrong file, (2) Data written using format statement, (3) Data written in the wrong format, (4) Data written with wrong carriage control, (5) Incomplete or missing output, (6) Output field size too small, (7) Line count and page eject problems.
- F. Examples of INTERFACE ERRORS include: (1) Wrong subroutine called, (2) Call to subroutine made in wrong place, (3) Subroutine arguments not consistent in type, units, order, etc, (4) Subroutine called is nonexistent.
- G. Examples of ARRAY PROCESSING ERRORS include: (1) Array not properly dimensioned, (2) Array referenced out of bounds, (3) Array being referenced at incorrect location, (4) Array pointers not incremented properly.
- H. Examples of DATA BASE ERRORS include: (1) Data should have been initialized in data base but wasn't, (2) Data initialized to incorrect value in data base, (3) Data base units are incorrect.
- I. Examples of OPERATION ERRORS include: (1) Operating system error, (2) Hardware error, (3) Operator error, (4) Test execution error.
- J. Examples of PROGRAM EXECUTION ERRORS include: (1) Time limit exceeded, (2) Core storage limit exceeded, (3) Output line limit exceeded, (4) Compilation error.
- K. Examples of DOCUMENTATION ERRORS include: (1) User manual error, (2) Interface spec error, (3) Design spec error, (4) Requirements spec error.
- L. Briefly describe the error(s).

APPENDIX B

RADC DATA BASE SCATTERPLOTS



PRODUCTIVITY VS DSLOC  
(MODERN PROGRAMS)



AD-A083 513

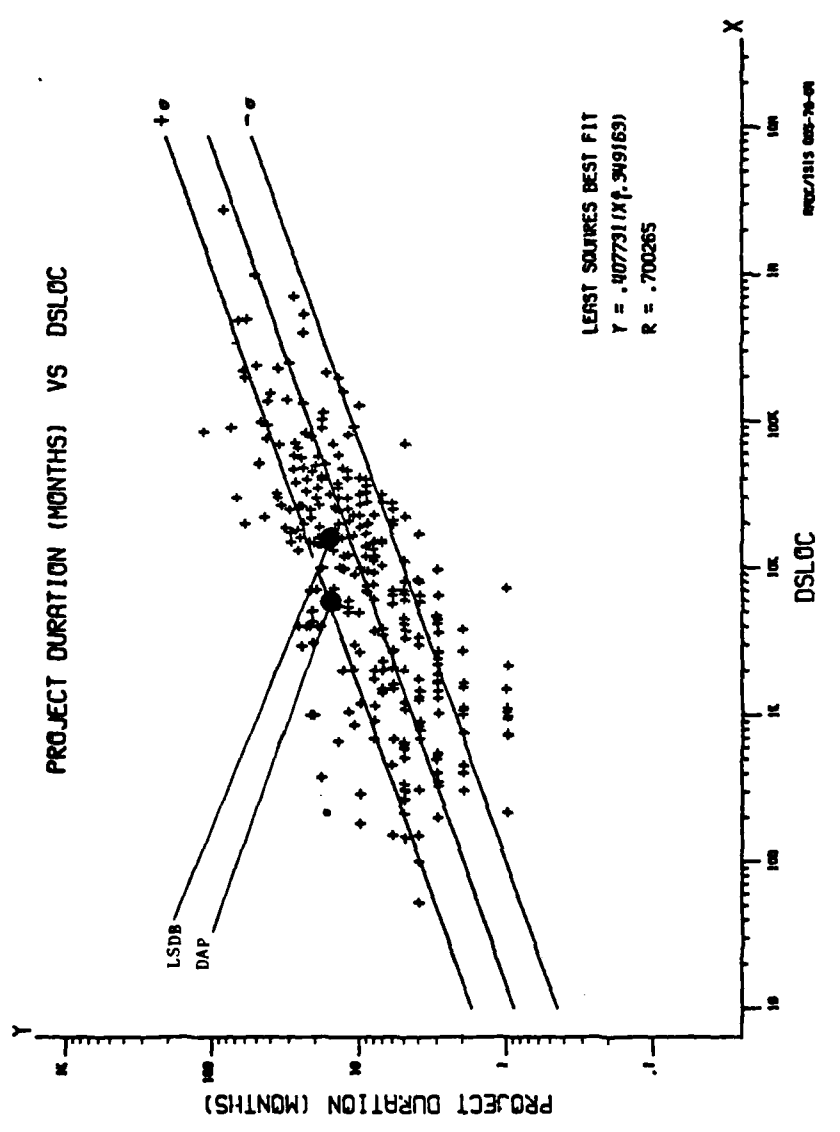
GENERAL ELECTRIC CO ARLINGTON VA F/8 9/2  
A MATCHED PROJECT EVALUATION OF MODERN PROGRAMMING PRACTICES. V--ETC(U)  
FEB 80 P WILLIMAN, B CURTIS F30602-77-C-0194  
791SP006 RADC-TR-80-6-VOL-2 NL

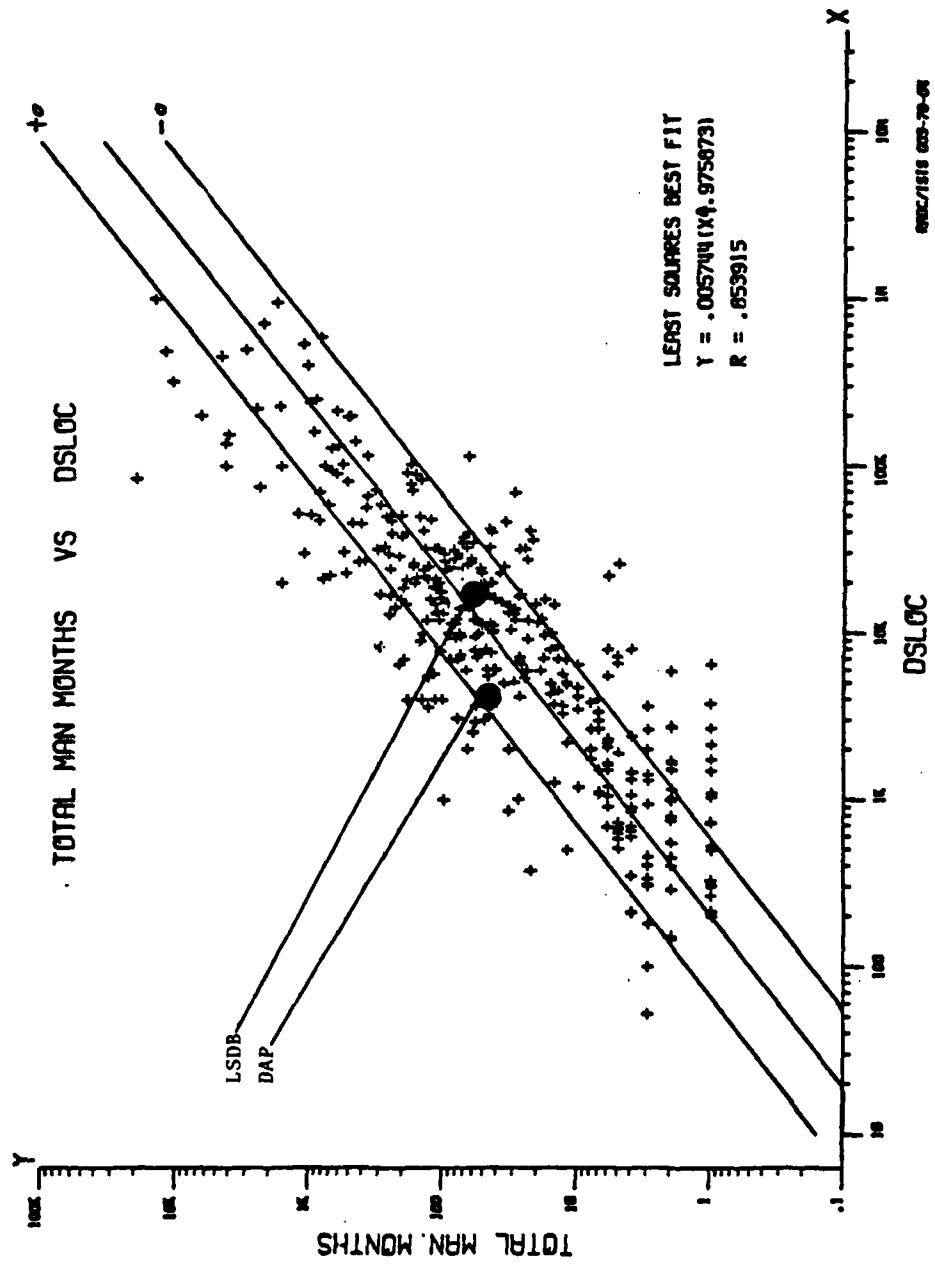
UNCLASSIFIED

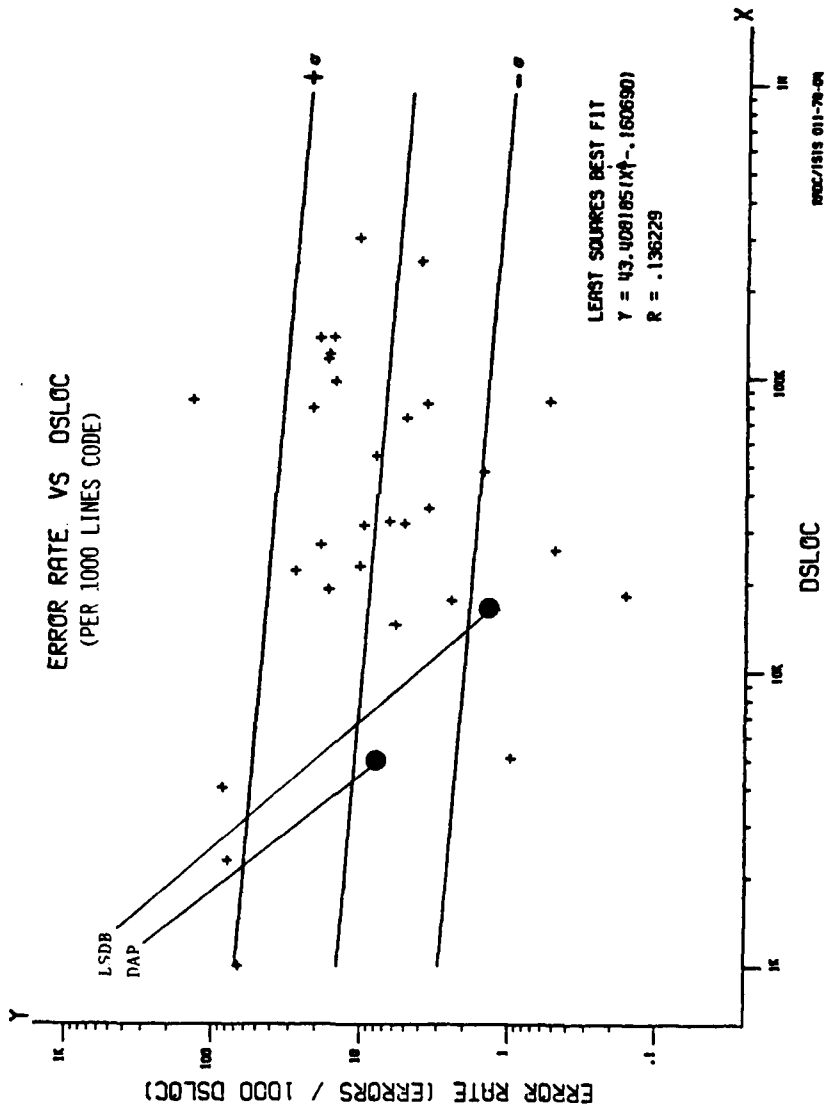
2 of 2  
AU  
ACR/PSA

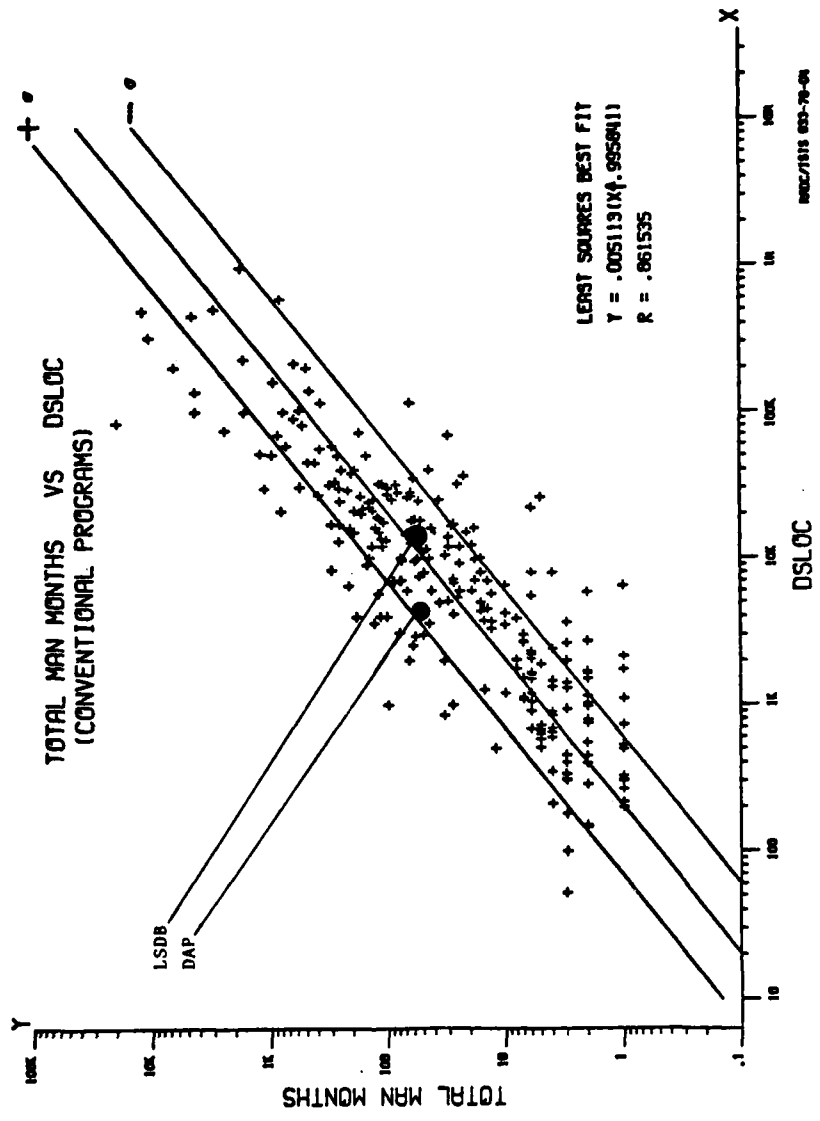


END  
DATE  
FILMED  
5 80  
DTIC

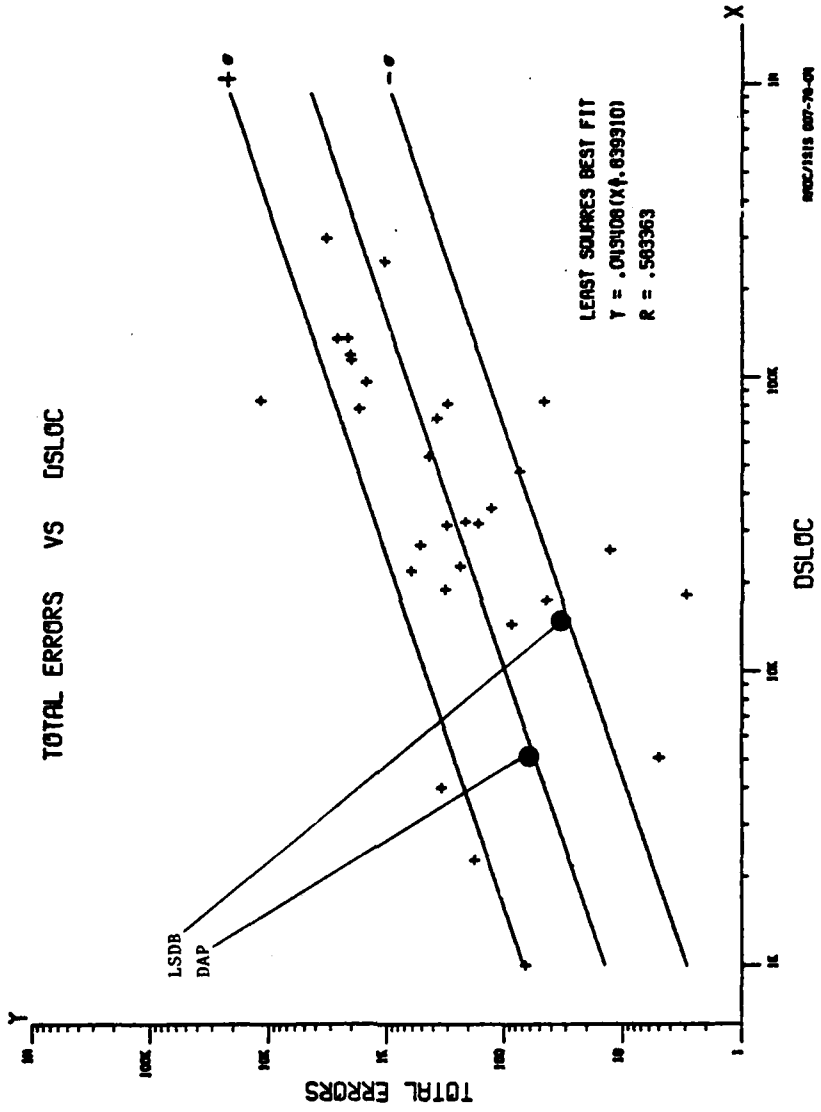


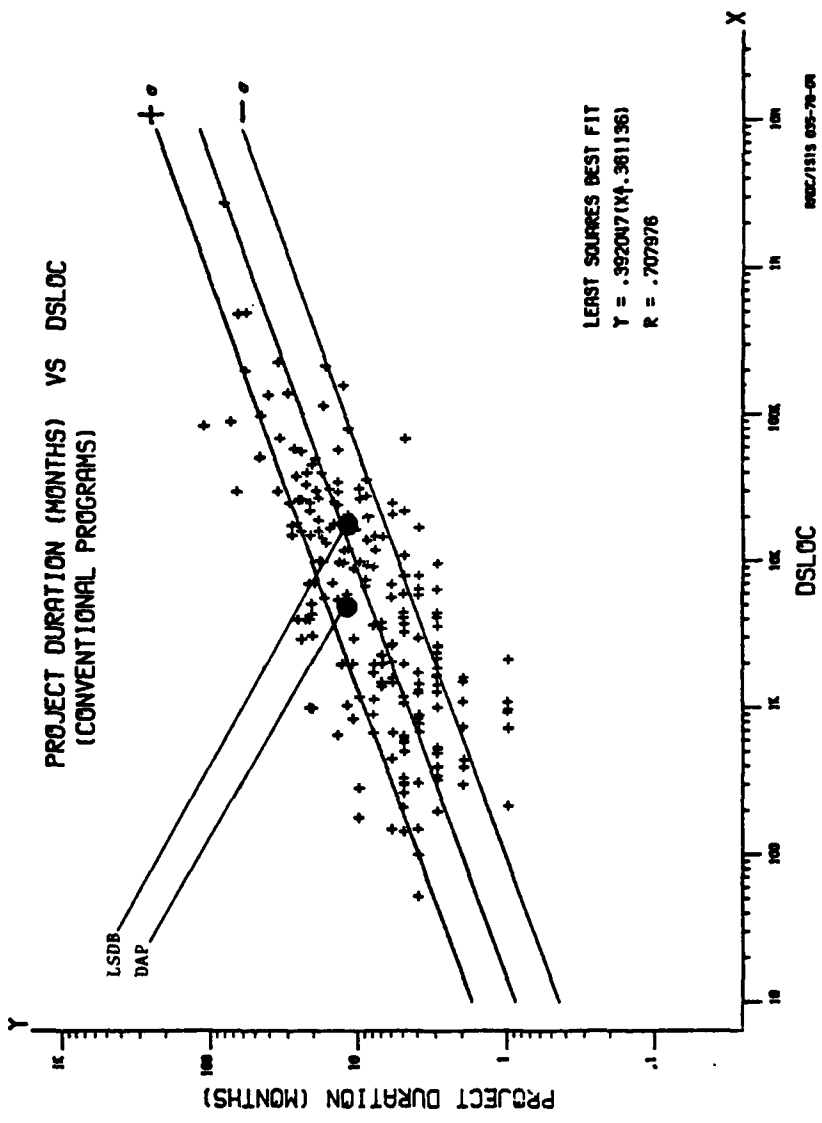




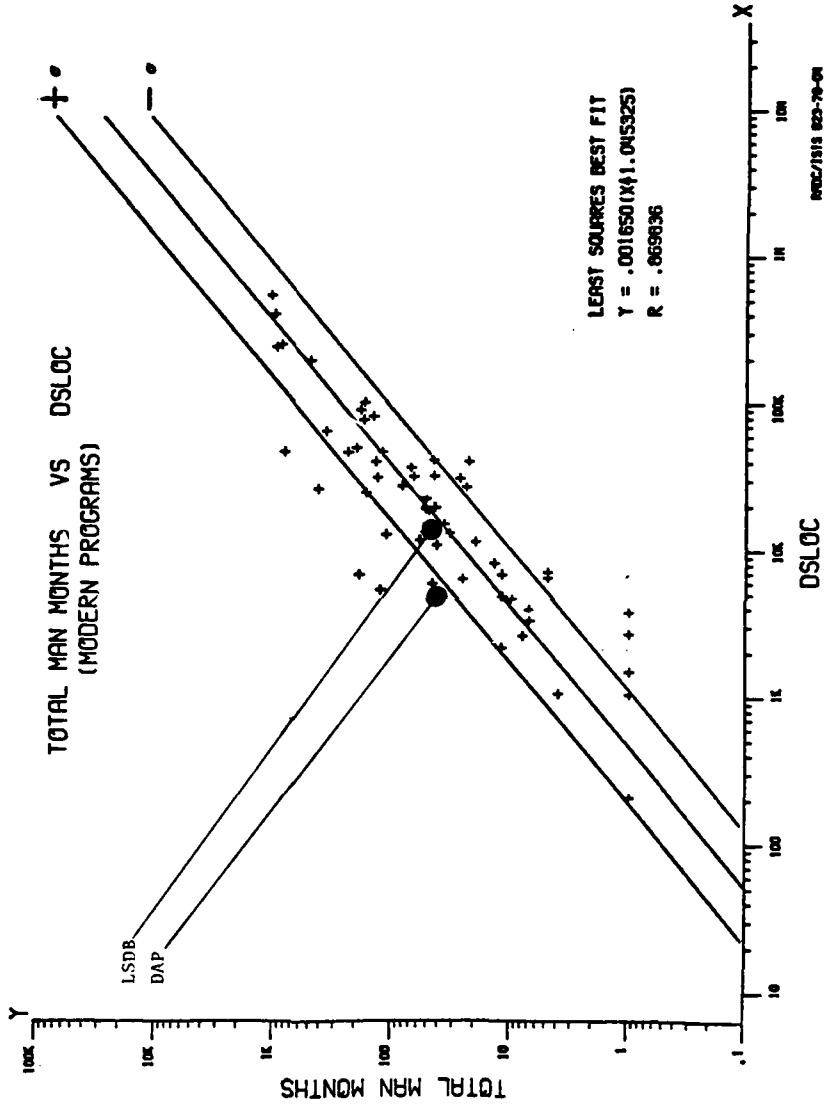


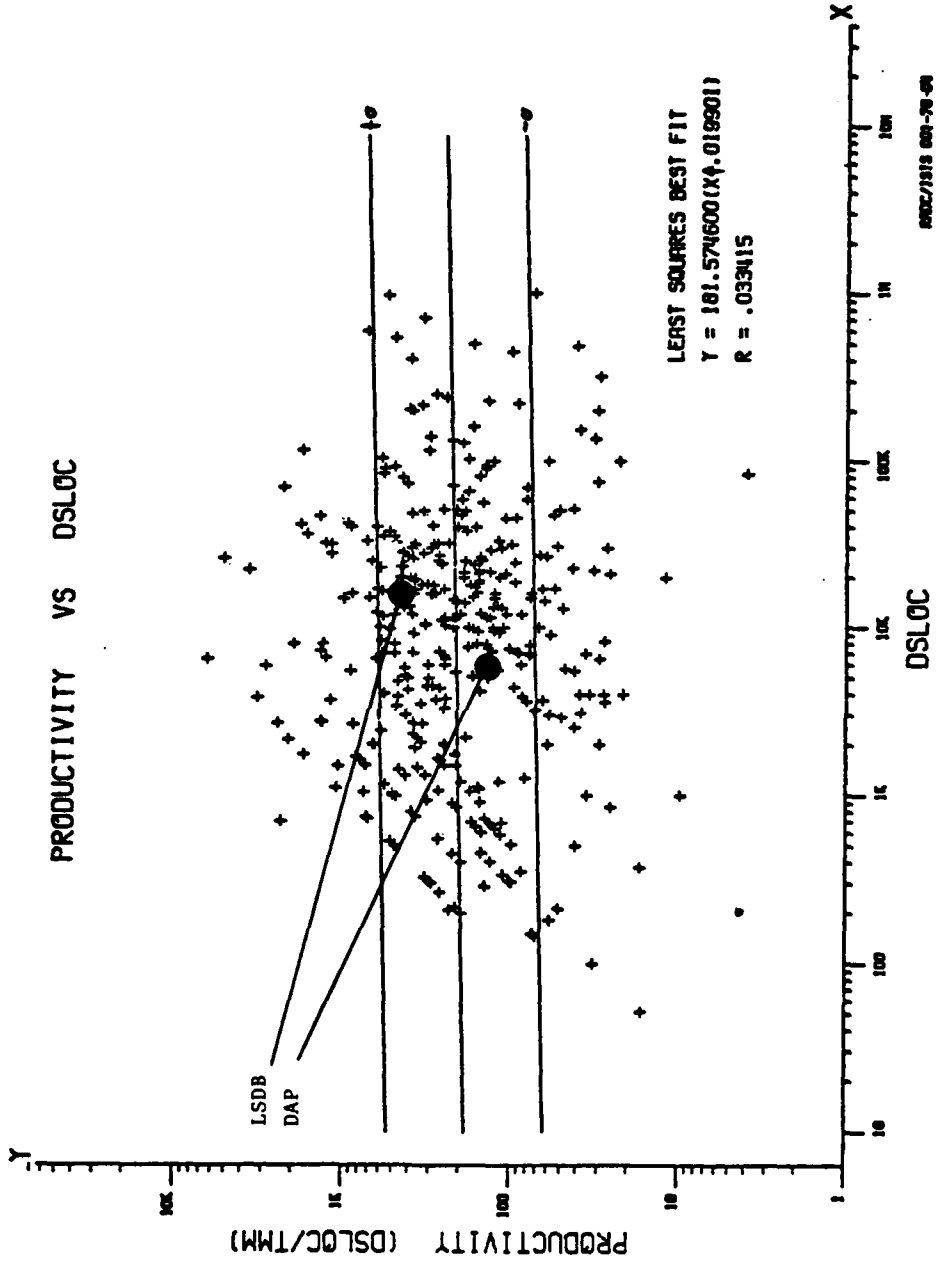
TOTAL ERRORS VS DSLOC

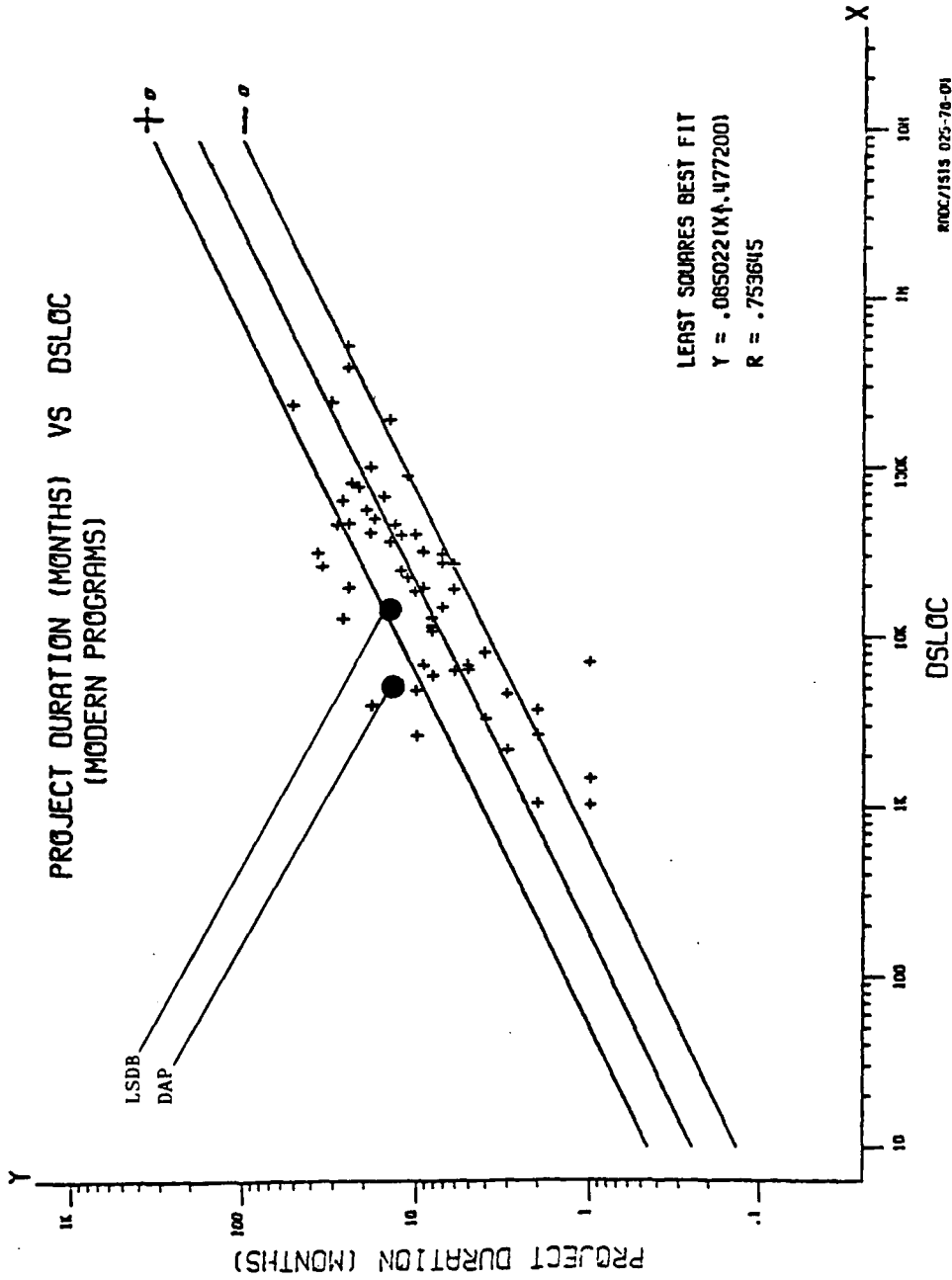


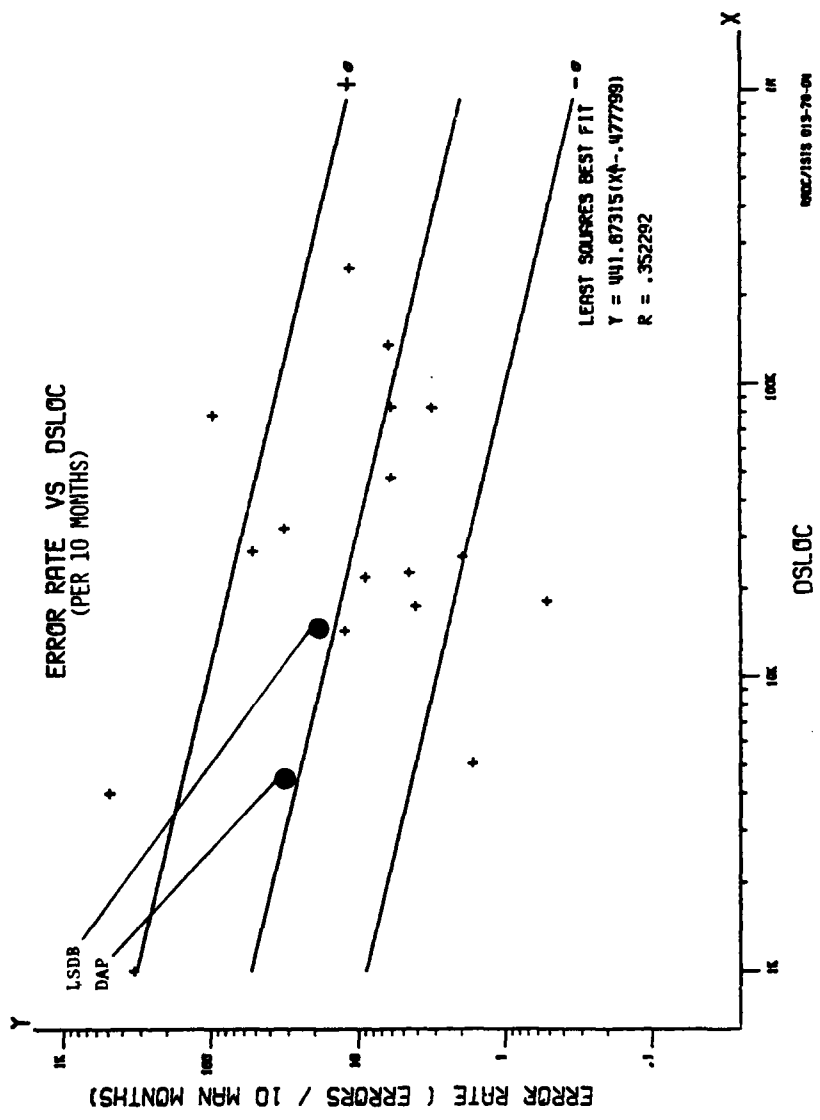




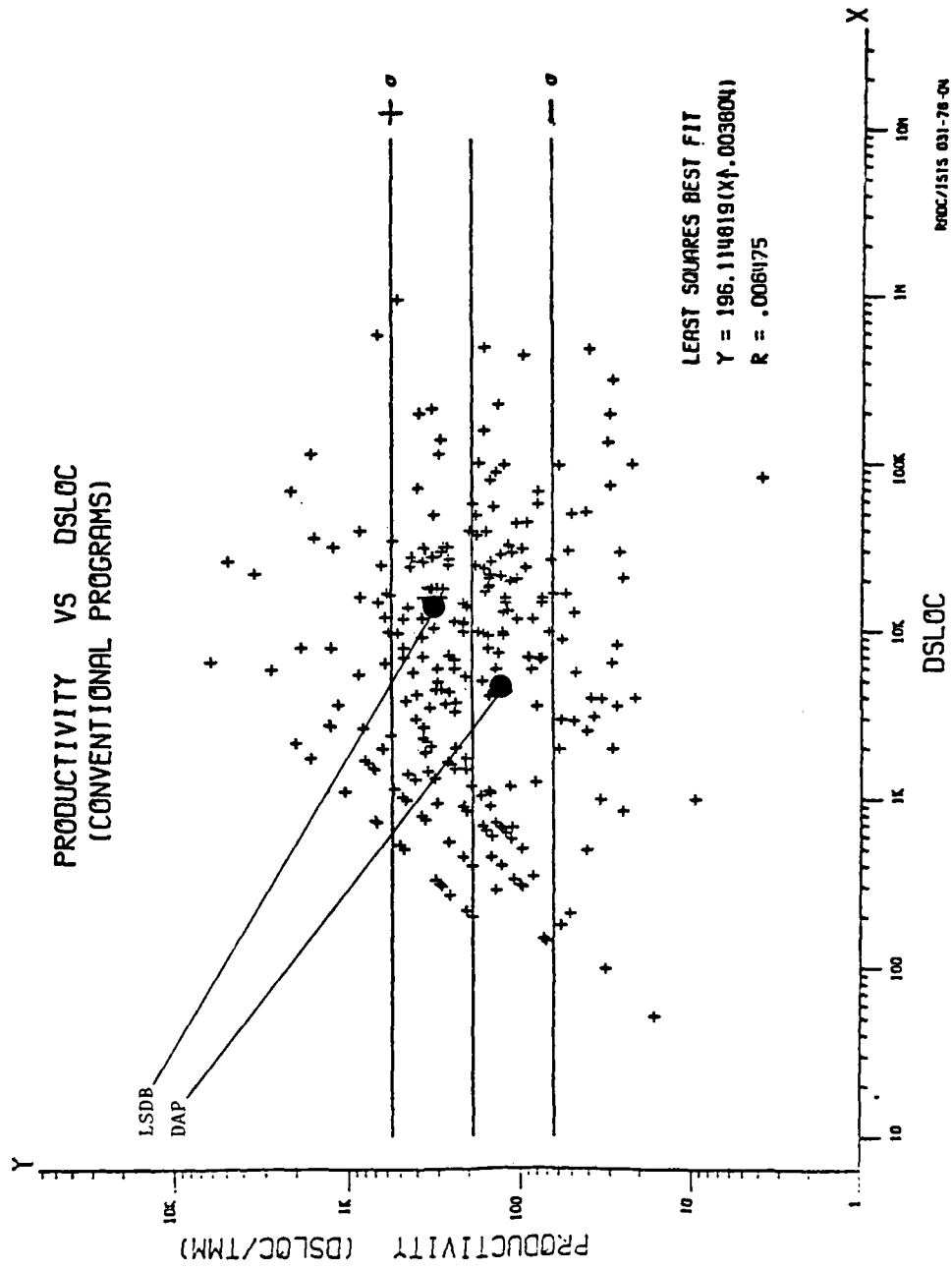


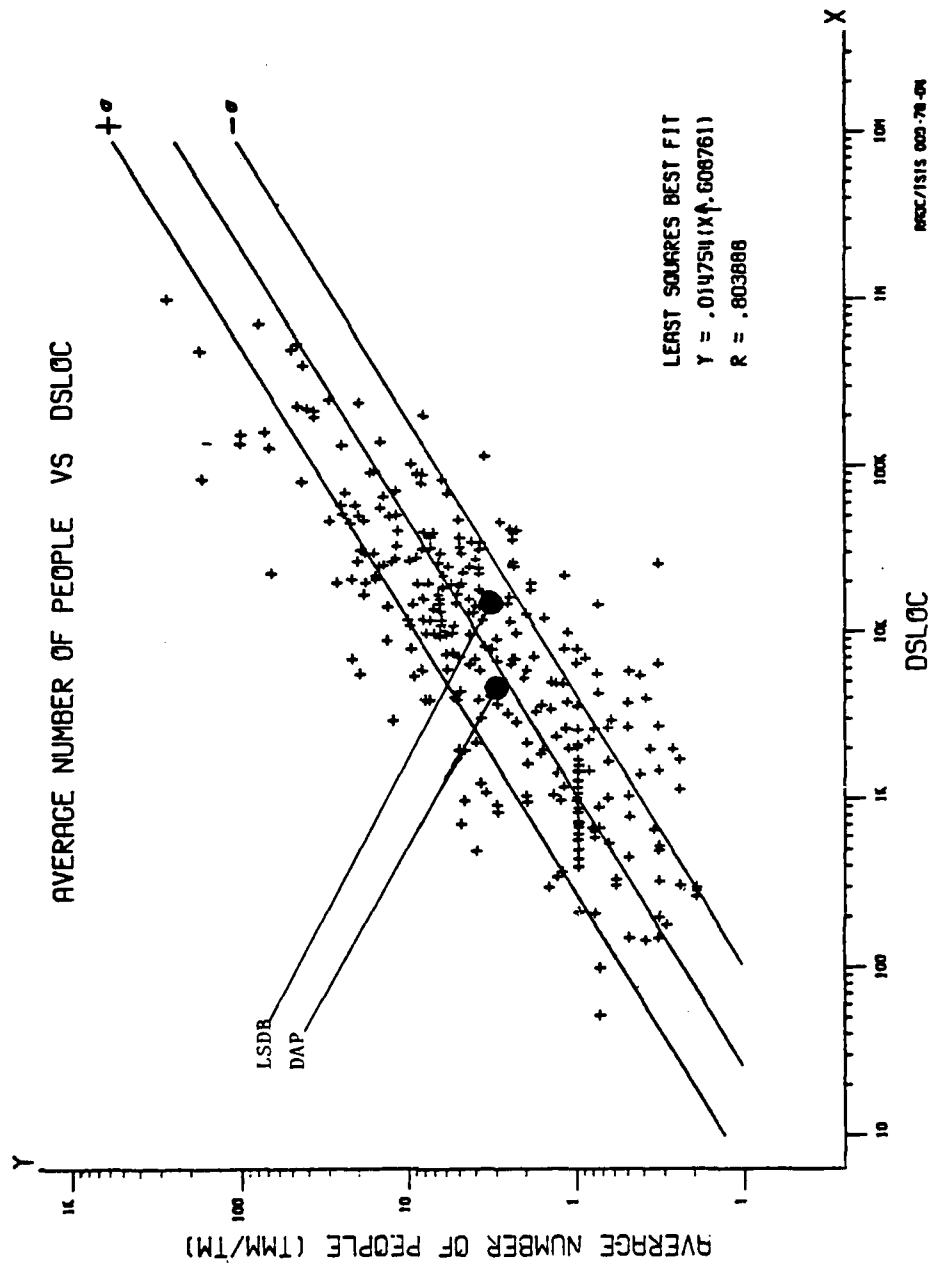






PRODUCTIVITY VS DSLOC  
(CONVENTIONAL PROGRAMS)





APPENDIX C

PROOF OF RELATIONSHIPS AMONG LEVELS OF  
PROGRAMMING SCOPE

Assumptions:

1. each subroutine has the same  $\eta$  and  $N$  as any other subroutine
2. furthermore - each  $\eta_1$  and  $\eta_2$  is  $1/2 \eta$  and each  $N_1$  and  $N_2$  is  $1/2 N$
3. we have subroutines in the subsystem

We intend to show for this highly simplified case that  $E_{UMAX}$  is greater than  $E_{LMAX}$  and both are expected to be greater than  $E_{MIN}$ .

First we derive the basic form of  $E_{MIN}$

$$\begin{aligned} E_{MIN} &= \frac{V_{MIN}}{\hat{L}_{MIN}} = S \left( \frac{N \log_2 \eta}{\left( \frac{2 \cdot \eta_2}{\eta_1 \cdot N_2} \right)} \right) \\ &= S \left( \frac{N \log_2 \eta}{\left( \frac{2 \cdot \frac{1}{2} \eta}{\frac{1}{2} \eta \cdot \frac{1}{2} N} \right)} \right) \\ &= S \frac{SN^2 \log_2 \eta}{4} \end{aligned}$$

PROOF 1: Establish that  $E_{MIN} < E_{LMAX}$

$$\begin{aligned} E_{LMAX} &= \frac{V_{MAX}}{\hat{L}_{MAX}} = \left( \frac{SN \log_2 \eta}{\left( \frac{2 \cdot \eta_2}{\eta_1 \cdot S \cdot N_2} \right)} \right) \\ &= \frac{SN \log_2 \eta}{\left( \frac{2 \cdot \frac{1}{2} \eta}{\frac{1}{2} \eta \cdot S \cdot \frac{1}{2} N} \right)} \end{aligned}$$



$$= \frac{S^2 N^2 \log_2 n}{4}$$

$$= S \cdot E_{\text{MIN}}$$

$$\therefore E_{\text{MIN}} < E_{\text{LMAX}} \text{ for } S > 1.$$

PROOF 2: Establish that  $E_{\text{UMAX}} > E_{\text{LMAX}}$

$$\begin{aligned} E_{\text{UMAX}} &= \frac{V_{\text{UMAX}}}{L_{\text{UMAX}}} = \frac{SN \log_2 (Sn)}{\left( \frac{2 \cdot Sn_2}{Sn_1 \cdot SN_2} \right)} \\ &= \frac{SN (\log_2 S + \log_2 n)}{\left( \frac{2 \cdot S \cdot \frac{1}{2} n}{S \cdot \frac{1}{2} N} \right)} \\ &= \frac{SN (\log_2 S + \log_2 n)}{\left( \frac{4}{SN} \right)} \\ &= \frac{S^2 N^2 \log_2 S + S^2 N^2 \log_2 n}{4} \end{aligned}$$

$\left. \begin{array}{l} \text{Note that } E_{\text{LMAX}} = \frac{S^2 N^2 \log_2 n}{4} \\ \text{and replace into equation for } E_{\text{UMAX}} \end{array} \right\}$

$$= \frac{S^2 N^2 \log_2 S + E_{\text{LMAX}}}{4}$$

$$\therefore E_{\text{LMAX}} < E_{\text{UMAX}}$$

and by transitivity  $E_{\text{MIN}} < E_{\text{UMAX}}$

Q.E.D.

APPENDIX D

SOFTWARE QUALITY METRICS

Comparisons of LSDB and DAP on McCall's Quality Metrics

Metric	LSDB		DAP	
SI.2 USE OF STRUCTURED LANGUAGE OR PREPROCESSOR Structured language or structured language preprocessor used to implement module. If used = 1, if not used = 0.	1	1	0	0
SI.4 MEASURE OF CODING SIMPLICITY (by module)				
(1) Module flow top to bottom	.43	.75	0	0
(2) Negative Boolean or complicated compound Boolean expressions used. $\left(1 - \frac{\# \text{ of above}}{\# \text{ executable statements}}\right)$	.97	.96	.98	.95
(3) Jumps in and out of loops. $\left(\frac{\# \text{ single entry/single exit loops}}{\text{total \# loops}}\right)$	.13	.69	.45	.70
(4) Loop index modified. $\left(1 - \frac{\# \text{ loop indices modified}}{\text{total \# loops}}\right)$	1	1	1	1
(5) Module is not self-modifying	1	1	1	1
(6) All arguments passed to a module are parametric.	0	0	0	0
(7) Number of statement labels. $\left(1 - \frac{\# \text{ labels}}{\# \text{ executable statements}}\right)$	1	1	.83	.74
(8) Unique names for variables.	0	0	0	0
(9) Single use of variables.	1	1	1	1
(10) No mixed mode expressions.	1	1	1	1
(11) Nesting level. $\left(\frac{1}{\text{max nesting level}}\right)$	.20	.17	.17	.17
(12) Number of branches. $\left(1 - \frac{\# \text{ branches}}{\# \text{ executable statements}}\right)$	.66	.26	.62	.33
(13) Number of GOTOs. $\left(1 - \frac{\# \text{ GOTO statements}}{\# \text{ executable statements}}\right)$	1	1	.76	.72
(14) No extraneous code exits.	1	1	1	1

(Cont'd)

Metric	LSDB	DAP		
(15) Variable mix in a module. $\left( \frac{\# \text{ internal variables}}{\text{total } \# \text{ variables}} \right)$	.73	.30		
(16) Variable density. $\left( 1 - \frac{\# \text{ variables}}{\# \text{ exec statements}} \right)$	.46	.13		
SUBROUTINE METRIC VALUE	.67	.69	.63	.57
SD.1 QUANTITY OF COMMENTS (by module) $\left( \frac{\# \text{ of comments (nonblank)}}{\text{Total } \# \text{ lines (nonblank)}} \right)$	.49	.28	.46	.17
SD.2 EFFECTIVENESS OF COMMENTS MEASURE				
(1) Modules have standard formatted prologue comments which describe: - Module name/version number - Author - Date - Purpose - Inputs - Outputs - Function - Assumptions - Limitations and restrictions - Accuracy requirements - Error recovery procedures - References $\left( 1 - \frac{\# \text{ modules violate rule}}{\text{total } \# \text{ modules}} \right)$	0	0	0	0
(2) Comments set off from code in uniform manner. $\left( 1 - \frac{\# \text{ modules violate rule}}{\text{total } \# \text{ modules}} \right)$	0	0	0	1
(3) All transfers of control & destinations commented. $\left( 1 - \frac{\# \text{ modules violate rule}}{\text{total } \# \text{ modules}} \right)$	.14	.25	0	0

(Cont'd)

Metric	LSDB	DAP		
(4) All machine dependent code commented. $1 - \left( \frac{\# \text{ modules violate rule}}{\text{total \# modules}} \right)$	1	N/A	1	N/A
(5) All non-standard HOL statements commented. $1 - \left( \frac{\# \text{ modules violate rule}}{\text{total \# module}} \right)$	1	N/A	1	N/A
(6) Attributes of all declared variables commented. $1 - \left( \frac{\# \text{ modules violate rule}}{\text{total \# modules}} \right)$	0	1	0	0
(7) Comments do not just repeat operation described in language. $1 - \left( \frac{\# \text{ modules violate rule}}{\text{total \# modules}} \right)$	1	1	1	1
SUBROUTINE METRIC VALUE = Total scores from $\frac{\text{applicable elements}}{\# \text{ applicable elements}}$	.45	.45	.43	.40

SD.3 DESCRIPTIVENESS OF IMPLEMENTATION  
LANGUAGE MEASURE

(1) High order language used. $1 - \left( \frac{\# \text{ modules with direct code}}{\text{total \# modules}} \right)$	1	1	1	1
(2) Standard format for organization of modules followed. $1 - \left( \frac{\# \text{ modules violate rule}}{\text{total \# modules}} \right)$	1	1	1	1
(3) Variable names (mnemonic) descriptive of physical or functional property represented. $1 - \left( \frac{\# \text{ modules violate rule}}{\text{total \# modules}} \right)$	1	1	1	1
(4) Source code logically blocked and indented. $1 - \left( \frac{\# \text{ modules violate rule}}{\text{total \# modules}} \right)$	1	1	0	1

(Cont'd)

Metric	LSDB		DAP	
(5) One statement per line. $1 - \left( \frac{\# \text{ continuations} + \text{multiple}}{\text{statement lines}} \right)$ $1 - \left( \frac{\# \text{ continuations} + \text{multiple}}{\text{total \# lines}} \right)$	1	1	1	.99
(6) No language keywords used as names. $1 - \left( \frac{\# \text{ modules violate rule}}{\text{total \# modules}} \right)$	1	.88	0	1
SUBROUTINE METRIC VALUE = $\frac{\text{Total Score from applicable elements}}{\# \text{ applicable elements}}$	1.00	.98	.83	.83

MO.2 MODULAR IMPLEMENTATION MEASURE

(1) Hierarchical structure. $1 - \left( \frac{\# \text{ violations of hierarchy}}{\text{total \# modules}} \right)$	.43	.25	N/A	N/A
(2) All modules do not exceed standard module size (100). $1 - \left( \frac{\# \text{ modules} > 100}{\text{total \# modules}} \right)$	1	1	0	N/A
(3) All modules represent one function. $1 - \left( \frac{\# \text{ modules violate rule}}{\text{total \# modules}} \right)$	1	1	0	N/A
(4) Controlling parameters defined by calling module. $1 - \left( \frac{\# \text{ modules violate rule}}{\text{total \# modules}} \right)$	0	0	N/A	N/A
(5) Input data controlled by calling module. $1 - \left( \frac{\# \text{ modules violate rule}}{\text{total \# modules}} \right)$	0	0	N/A	N/A
(6) Output data provided to calling module. $1 - \left( \frac{\# \text{ modules violate rule}}{\text{total \# modules}} \right)$	0	0	N/A	N/A

(Cont'd)

Metric	LSDB		DAP	
(7) Control returned to calling module. $1 - \left( \frac{\# \text{ modules violate rule}}{\text{total \# modules}} \right)$	1	1	N/A	N/A
(8) Modules do not share temporary storage.	1	0	N/A	N/A
SUBROUTINE METRIC VALUE = $\frac{\text{Total score from applicable elements}}{\# \text{ applicable elements}}$	.55	.41	0	0

APPENDIX E

MANHOURS BY MONTH AND WORK BREAKDOWN STRUCTURE FOR  
LSDB AND DAP



LAUNCH SUPPORT DATA BASE - SUMMARY OF CHARGES: MAY 76 TO JUN 77 (MAN-HOURS)

WORK BREAKDOWN STRUCTURE	MAY 1976	JUN 1976	JUL 1976	AUG 1976	SEP 1976	OCT 1976	NOV 1976	DEC 1976	JAN 1977	FEB 1977	MAR 1977	APR 1977	MAY 1977	JUN 1977	TOTAL
REQUIREMENT ALLOCATION															
DEVELOPMENT SPECIFICATION	31	36													67
ION UPDATE	112	252	48												412
PRELIMINARY DESIGN	80	172													252
DT&E TEST PLAN	152	216	88												456
PCR AGENDA/MATERIAL	40	36													76
PCR MINUTES		16													16
DETAIL DESIGN			424	327	224	293	158	190							1616
CDR AGENDA/MATERIAL					149										149
CDR MINUTES				241	72										313
CDR/INTEGRATION						76	220	304	392	344	188				1524
DT&E TESTING			80	320	296		72	208	100		152	216	281	216	856
DT&E TEST PROCEDURES															1076
FCA AGENDA/MATERIAL															
DT&E TEST REPORT													50	11	61
FCA															
FCA MINUTES															
PRODUCT DEFINITION															
TRAINING PLANS								80							80
TRAINING AIDS															
DRAFT USER MANUAL								64	140	166	64				434
TRAINING CERTIFICATION															
FCA AGENDA															
USER MANUAL															
PRODUCT SPECIFICATION											80	276	148	84	148
VERSION DESCRIPTION DCC.													96		536
FCA															
FCA MINUTES															
TOTAL BY MONTH:	415	728	640	888	741	369	450	846	632	510	484	492	575	311	8081

DATA PROCESSING SUMMARY OF CHARGES TO DATE BY WRS # AND MONTH (MANHOURS)

WORK BREAKDOWN STRUCTURE	OCT 1976	NOV 1976	DEC 1976	JAN 1977	FEB 1977	MAR 1977	APR 1977	MAY 1977	JUN 1977	JUL 1977	AUG 1977	SEP 1977	OCT 1977	NOV 1977	TOTAL
REQUIREMENT ALLOCATION DEVELOPMENT SPECIFICATION ICM UPDATE	436	155	322	112							352	32			1409
PRELIMINARY DESIGN															
DTGE TEST PLAN				80	208										208
PDR AGENDA/MATERIAL					132										212
PDR MINUTES					12										12
DETAILED DESIGN						272	100								372
CWR AGENDA/MATERIAL						40									40
CWR MINUTES							6								6
CWR MINUTES							10								10
CONF/INTEGRATION							238	256	267	292	40	120	104	32	1349
DTGE TESTING								54	65	324	445	372	241	389	1771
DTGE TEST PROCEDURES															269
FCA AGENDA/MATERIAL															
DTGE TEST REPORT															
FCA											20	24	104		148
FCA MINUTES															
PRODUCT DEFINITION							10								10
TRAINING PLANS															450
TRAINING AIDS															20
DRAFT USER MANUAL								134	216	100	20				
TRAINING CERTIFICATION															
FCA AGENDA															
USERS MANUAL										60	120	60		108	188
PRODUCT SPECIFICATION														62	308
VERSION DESCRIPTION DOC.															
FCA															
FCA MINUTES															
TOTAL BY MONTH	436	155	322	192	352	312	364	444	548	926	1023	668	449	591	6782

APPENDIX F

NUMBER OF ERRORS BY CATEGORY AND MONTH  
FOR LSDB

ERROR CATEGORY	MAR 1976	APR 1976	MAY 1976	JUN 1976	JUL 1976	AUG 1976	SEP 1976	OCT 1976	NOV 1976	DEC 1976	JAN 1977	FEB 1977	MAR 1977	APR 1977	MAY 1977	JUN 1977	UNDATED	TOTAL ERRORS
ALGORITHMIC:																		
COMPUTATIONAL			2	1			2				14	7	3	2	4			5
LOGIC		9	6	2	5	13	4	7	15	7								98
DATA INPUT		3	3															17
DATA HANDLING			11						11					2				12
DATA OUTPUT			1															3
INTERFACE		1																0
ARRAY PROCESSING		1																1
DATABASE		2		1						1								1
NON-ALGORITHMIC:																		4
OPERATION		2	1	1	2	4	4	16	21	23	18	12	4	6	5	1		115
PROGRAM EXECUTION		3	2	1	1	1	1	3	3	3	3	10	2	5	5			41
DOCUMENTATION																		0
KEYPUNCH		2	3	2	7	7	1	1	2	3	5	5	6	2	3	1		51
JOB CONTROL LANGUAGE		2	11	1	5	4	2	19	2	6	5	1	6	1	4			73
UNCLASSIFIED		3	3	1	4	1	6	2	7	3	18	22	9	5	2	2		88
TOTAL ERRORS	13	37	37	17	11	30	19	48	58	42	64	57	30	23	18	4		508
TOTAL RUNS	57	177	147	118	72	117	119	160	248	174	283	361	291	231	112	50	2	2719

*MISSION*  
*of*  
*Rome Air Development Center*

*RADC plans and executes research, development, test and selected acquisition programs in support of Command, Control Communications and Intelligence (C<sup>3</sup>I) activities. Technical and engineering support within areas of technical competence is provided to ESD Program Offices (POs) and other ESD elements. The principal technical mission areas are communications, electromagnetic guidance and control, surveillance of ground and aerospace objects, intelligence data collection and handling, information system technology, ionospheric propagation, solid state sciences, microwave physics and electronic reliability, maintainability and compatibility.*