7N
(1+5)

PROCEEDINGS FROM THE THIRD SUMMER SOFTWARE ENGINEERING WORKSHOP

HELD ON

SEPTEMBER 18, 1978

AT

GODDARD SPACE FLIGHT CENTER
GREENBELT, MARYLAND

NASA
National Aeronautics and
Space Administration

PROCEEDINGS

OF

THIRD SUMMER SOFTWARE ENGINEERING WORKSHOP

Organized By:

Software Engineering Laboratory
GSFC

September 18, 1978

GODDARD SPACE FLIGHT CENTER
Greenbelt, Md.

SCHEDULE FOR SOFTWARE ENGINEERING WORKSHOP

8:45 am Introduction – F. E. McGarry, GSFC

9:00 am Panel #1 – 'The Data Collection Process'

Chairperson:	Gerry Page, Computer Sciences Corporation
Member 1	David Weiss, Naval Research Lab
Member 2	Bill Curtis, General Electric
Member 3	Pat Ryan, SAI

10:30 am Coffee Break

10:45 am Panel #2 – 'Validation and Verification of Software Development Models'

Chairperson:	Vic Basili, University of Maryland
Member 1	Larry Putnam, Quantitative Software Measurement
Member 2	Sylvia Sheppard, General Electric
Member 3	Doug Brooks, IBM Corporation

12:15 pm Lunch

1:15 pm Panel #3 – 'Measuring Software Development Methodologies'

Chairperson:	Marv Zelkowitz, University of Maryland
Member 1	Bob Reiter, University of Maryland
Member 2	Phil Milliman, General Electric
Member 3	Paul Scheffer, Martin Marietta Corporation

2:45 pm Coffee Break

3:00 pm Panel # 4 – 'Current Activities and Future Direction'

Chairperson:	Frank McGarry, GSFC
Member 1	Lorraine Duval, IITRI
Member 2	Vic Basili, University of Maryland
Member 3	Chuck Everhart, Brown Engineering

4:30 pm Adjourn

PANEL #1

THE DATA COLLECTION PROCESS

Chairperson	Gerry Page (Computer Sciences Corp)
Member #1	David Weiss (Naval Research Lab)
Member #2	Bill Curtis (General Electric)
Member #3	Pat Ryan (Science Applications Inc)

DATA COLLECTION FOR THE SOFTWARE ENGINEERING LABORATORY

David M. Weiss
Naval Research Laboratory

The Software Engineering Laboratory (SEL) is a project started and sponsored by Goddard Space Flight Center (GSFC) to study software development. The purpose of the SEL is to find ways to improve the software developed for and by GSFC. The approach used is to collect data for analysis by monitoring current software development projects. Analysis of the data is expected to provide insight into such issues as

- What resources are used for different types of projects and during different stages of a project?
- What activities are in progress during different project stages, and how much effort is expended on those activities?
- What kinds of changes are made to software as it is being developed, and how are those changes distributed over the development cycle?
- How good are current software cost estimation techniques?
- How many and what kinds of errors are committed during the software development cycle?

The preceding and other, similar, analyses will be performed for different software development methodologies. The results of these analyses are expected to provide the basis for deciding which methodologies yield better software.

METHODOLOGY

Data collection and analysis for the SEL is a cooperative project among Computer Sciences Corporation (CSC), GSFC, and the University of Maryland (UM). Data is collected in parallel with software development. The data collection instruments are a series of six different forms that focus on the following areas:

- Definition of the software development methodology used during the project,
- Estimation of manpower and computer resources needed to complete the project,
- Resources actually used during the project, on a weekly basis,
- Activities (e.g. design, coding, reviews, etc.) occurring during the development,
- Changes made and errors committed during the development cycle.

FORMS

The forms used by the SEL were developed over a period of about a year and a half. The first step was to decide on a list of questions of interest to be answered by the study. A set of forms was then designed to capture the data needed to answer those questions. This process was repeated several times. The forms were then used on a few projects, and revised. The forms currently in use are described below.

At the beginning of each project a General Project Summary form is completed providing estimates of the resources needed and a definition of the methodology to be used on the project. The General Project Summary is updated midway through the project and at the completion of the project. Each time a new component of the system under development is identified, a Component Summary Form is completed describing the component, giving the reason(s) for its existence and, the resources needed to complete it. The Component Summary form is also updated partway through the project and at the end of the project. On a weekly basis, Resource Summary and Component Status Report forms are completed. The Resource Summary provides, for each person working on the project, the amount of time spent on and the computer usage for the project. The Component Status Report gives, for each programmer, for each component of the project, the amount of time spent in design, development, testing, and other activities. For each change made to the software, a Change Report form is completed describing the change. The type of change, the reason for the change, the amount of time required to make the change, and, for error corrections, the source of the error, the techniques used for detecting and correcting the error, and the project stage at which the error was introduced are included on the Change Report form. Finally, for each computer run made during the project, a Computer Program Run Analysis form is completed, containing the duration, purpose, and results of the run.

DATA PROCESSING

All data collection forms, except for the General Project Summary, once completed, are initially manually verified at CSC. They are then sent to GSFC to be encoded for computer entry. Once encoded, the data are entered into a PDP-11 computer, validated by a program, and written onto a tape. The tape is then sent to UM, the data revalidated and entered into a data base using the Ingres data base system on another PDP-11. The data are then analyzed by a set of programs designed to calculate various parameters of interest.

Because data collection proceeds in parallel with system development, the SEL data collection and processing procedures offer excellent opportunities for ensuring data accuracy. As an example, part of the procedure for processing Change Reports allows the reports to be examined by a UM researcher shortly after they are received at GSFC. If a form is incomplete or inconsistent in some way, the researcher can then contact the programmer who completed the form to correct the information on it.

DATA COLLECTION PROBLEMS

The major problem in collecting and processing SEL data is ensuring the completeness and accuracy of the data from the time it is collected to the time it is analyzed. Errors are introduced at the time

the forms are filled out, at the time they are encoded for computer entry, and at the time the encoded data are keypunched. Many of the keypunching and encoding errors are detected by the validation programs. Errors made in filling out a form often are only detected when an SEL researcher reviews the form. Correcting these errors usually requires that the person who completed the form be contacted.

An early problem in processing forms was the volume of data involved. The number of people needed for encoding and computer entry of the data was significantly underestimated. There are currently three people assigned to this task full-time.

Other problems involved in data collection are training software development project personnel in completing forms, and retraining them when the forms are revised, keeping the overhead involved in filling out forms reasonably low so that it does not significantly interfere with project schedules, ensuring that the data requested on the forms is sufficient to answer questions of interest, and devising a bookkeeping system to keep track of the whereabouts of all forms from the time they are completed to the time they are entered in the data base.

GOALS OF THE SOFTWARE ENGINEERING LABORATORY

- Gain insight into software development process
  - What resources are used?
  - What activities are in progress at different stages?
  - How good are current estimation techniques?
  - What kinds of changes are made?
  - How many and what kinds of errors are committed?
- Compare different software development methodologies
  - Which methodologies produce "better" software?

METHODOLOGY OF SEL

- Cooperative effort among CSC, GSFC, UM
- Collect and analyze data
  - Methodology definition (General Project Summary, Component Summary)
  - Resource estimation (General Project Summary, Component Summary)
  - Resource usage (Resource Summary, Component Status, Computer Program Run Analysis)
  - Activities (Resource Summary, Component Status)
  - Changes and errors (Change Report)

DATA COLLECTION

1. Programmers and managers complete forms
2. Forms verified initially at CSC
3. Forms encoded for computer entry at GSFC
4. Encoded data checked by validation program at GSFC
5. Encoded data revalidated and entered into data base at UM

CHANGE REPORT FORM PROCESSING

1. Change form completed by programmer at each change
2. Manual completeness check made at CSC
3. Manual completeness and consistency check made by UM
4. Interview with programmer when necessary by UM
5. Encoding for computer entry by GSFC
6. Computer entry and validation at GSFC
7. Validation and data base entry at UM

SOME THEORETICAL PRESPECTIVES ON DEVELOPING A SOFTWARE LIFE CYCLE DATA BASE

CNTS

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

In developing a software life cycle data base, we make the rather obvious assumption that research is not a scavenger hunt. Although software development efforts generate an enormous assortment of numbers, research is not an attempt "to salvage usable goods by rummaging through refuse or discards" (Webster's definition of "scavenge"). Rather, data collection for research purposes should be designed from a theoretical model of the phenomena to be studied. It is important to identify the data to be collected from the factors in the model, rather than contorting the model to fit the data that happen to be lying around. The quality of the resulting data base may also hinge on assigning an individual the responsibility of data collection and editing.

In identifying data relevant to each factor in a theoretical model, there are a number of important considerations. First, the data should be collected at the appropriate level of explanation. The following list represents three possible levels of explanation:

- Software development project
- Programming team
- Individual programmers

Data collected at the project level is not sufficient by itself to explain processes occurring at the level of the individual programmer. Thus, average lines of code per man-month at the project level is not an adequate criteria for investigating the productivity of individual programmers. Performance at the project level involves effort spent integrating the work of programmers and programming teams above and beyond the work initially expended by programmers in developing their code. In analyzing data across levels of explanation it is important to specify rules for aggregation which identify how the work of the parts is integrated into the whole.

Performance itself is an ambiguously used term. Rather than attempting to identify an ultimate criterion, the wise approach might be to identify multiple criteria at several different levels of explanation. Managers like to talk of meeting schedules within budgets, delivering high quality products. Jim McCall and Gene Walters of our Sunnyvale, CA office have identified myriad attributes constituting software quality such as reliability, maintainability, portability, efficiency, etc. Regarding criteria relevant to schedule and budget, one can collect machine records (runs, errors, changes, cpu time, etc.), personnel and payroll records (manpower loadings, labor costs, absenteeism, etc.), and managerial performance ratings. Identification of multiple criteria represents to

software development managers the tradeoffs they must frequently make between schedule, budget, and quality.

In software life cycle research it is important to distinguish between objective data which are direct measurements of the phenomena under consideration and subjective data which are reports by project participants. While subjective data are important, they should be collected along with rather then instead of objective data. To evaluate the effect of a modern programming practice solely on the basis of managerial reports is to introduce into the analysis the perspectives and biases of the managers which flavor their conclusions about the technique. A warm feeling in the tummy should be backed by analyses of objective data.

There are two primary research strategies which can be employed in testing hypotheses from theorectical models depending on the type of controls which can be exercised over the variables affecting performance. In a laboratory situation, experimental controls can be exercised by manipulating the independent variable(s), holding all other situational factors constant, and minimizing the effect of individual differences among participants by randomly assigning them to different conditions of the experiment. The strongest causal statements can usually be made from rigorously controlled experiments. On the other hand, research in field settings must usually reply on statistical controls to study the effects of different variables. Through the use of multivariate correlational methods such as structural equation models or time series analysis, underlying relationships can be teased from the data and different causal models can be compared to determine which is most consistent with the data at hand. In using either experimental or statistical controls, it is always important to identify both the factors which may moderate the relationships observed and the populations to which the results can be generalized.

The Software Management Research Unit at General Electric is currently conducting research projects using each type of control discussed above. In subsequent articles, Sylvia Sheppart will report on our experimental work for the Office of Naval Research on human factors in software engineering and software complexity metrics. Phil Milliman will report on our work for Rome Air Development Center evaluating the effects of modern programming practices on software development projects.

In summary we propose the following guidelines for software life cycle research:

- Begin with a theoretical model
- Identify an appropriate research strategy
- Appoint someone responsible for data collection
- Collect data which is
  - appropriate to the level of explanation
  - objective
  - longitudinal

- Identify multiple criteria
- Hire a good statistician

ACKNOWLEDGEMENT

Work from which this paper was drawn was performed pursuant to contracts #N0014-77-C-0158 with Engineering Psychology Programs, Office of Naval Research and #F30603-77-C-0194 with Rome Air Development Center. The views expressed in this paper, however, are not necessarily those of either the United States Navy or Air Force or the Department of Defense.

OMIT TO P.18

"THE DATA COLLECTION PROCESS"

Pat Ryan, SAI

Extensive error and production data has been collected on two software products developed in the Huntsville Office of SAI. Specifically, the following kind of data has been collected and is being tabulated.

- Modules vs. Number of Errors
- Personnel vs. Number of Errors
- Months vs. Number of Errors
- Number of Each Error Type
  (39) Possible Types)
- Modules vs. Number of Runs
- Modules vs. Total CPU Time
- System Builds vs. CPU Time
- System Builds vs. Number of Errors

On the basis of the above efforts we feel as though a productive data collection effort can be carried out on most projects as long as it is of reasonable overhead and non retributory to those participating in its collection. Further, we feel as though the collection process itself will raise the inherent quality of the work being monitored.

SOFTWARE SHOULD BE

- CORRECT – with respect to its requirements
- ROBUST – with respect to its inputs and modifications

SOFTWARE ERRORS

- In requirements, because of misunderstanding, poor judgement, on inconsistency
- In Design or Code, because of syntactics or poor judgements
- Inconsistencies between consecutive stages.

PROBLEM REPORT

1

PROBLEM REPORT NUMBER [2]	DATE [7]
TYPE OF CHANGE 1. Error 2. Improvement [15]	TYPE OF ERROR Use Error Categorization [16]
TYPE OF IMPROVEMENT 1. Delete Unnecessary Requirement 2. Additional Capabilities 3. Increased Efficiency 4. Analysis Aid 5. Other _____ [18]	IMPACT OF IMPROVEMENT 1. Major 2. Moderate 3. Minor [19]

2 ERROR/IMPROVEMENT DESCRIPTION [2] _____

3 MODULE(S) AFFECTED [2] _____

4 SOLUTION DATE [2] __/__/__

5 SOLUTION DESCRIPTION [2] _____

* *

REVIEW COMMITTEE USE ONLY

6

1. Should Be Reviewed 2. Should NOT Be Reviewed 3. Program Development Error Review [10]	DATE OF REVIEW [2]
REVIEW DECISION 1. Approved For Correction 2. Disapproved APPROVAL: _____ [11]	REASON FOR DISAPPROVAL 1. Hardware Problem 2. Duplication 3. High Cost 4. No Problem 5. Other _____ [12]

CORRECTION PERSONNEL ASSIGNED (Initials) [13] _____

7 SOLUTION REVIEW DATE [2] __/__/__ SOLUTION APPROVAL: _____

BASELINE VERSION UPDATE DATE [10] __/__/__

* *

* CLOSED OUT DATE [18] __/__/__ *

* APPROVAL: _____ *

* *

SYSTEM TEST REPORT

Module Names:

Testing Period: Start - / / Finish - / /

	CPU Time		Error Numbers	Comments
	min	sec		
1.				
2.				
3.				
4.				
5.				
6.				
7.				
8.				
9.				
10.				
11.				
12.				
13.				
14.				
15.				
16.				
17.				
18.				
19.				
20.				

PERSONNEL ACCEPTANCE

Data Collection Effort should be

- of minimal overhead effort to personnel
- non-retributive

PANEL #2

VALIDATION AND VERIFICATION OF SOFTWARE DEVELOPMENT MODELS

Chairperson	Vic Basili (University of Md.)
Member #1	Larry Putnam (Quantitative Software Measurement)
Member #2	Sylvia Sheppard (G.E.)
Member #3	Doug Brooks (IBM Corporation)

EXAMPLE OF AN EARLY SIZING, COST AND SCHEDULE ESTIMATE FOR AN APPLICATION SOFTWARE SYSTEM

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

ABSTRACT

Software development has been characterized by severe cost overruns, schedule slippages and an inability to size, cost and determine the development time early in the feasibility and functional design phases when investment decision must be made. Managers want answers to the following questions: Can I do it? How much will it cost? How long will it take? How many people? What's the risk? What's the trade-off? This portion of the paper shows how to size the project in source statements (S_s), how to relate the size to the management parameters (life cycle effort (K) and development time (t_d)] and the state-of-technology (C_k) being applied to the problem through the software equation, $S_s = C_k K^{1\backslash 3} t_d^{4\backslash 3}$. The software equation is then solved using a constraint relationship $K = |\nabla D| t_d^3$, where $|\nabla D|$ is the magnitude of the difficulty gradient empirically found to be related to system development characteristics measuring the degree of concurrency of major task accomplishment. Monte Carlo simulation is used to generate statistics on variability of the effort and development time. The standard deviations are used to make risk profiles. Finally, having the effort and development time parameters, the Rayleigh/Norden equation is used to generate the manpower and cash flow rate at any point in the life cycle. The results obtained demonstrate that engineering quality quantitative answers to the management questions can be obtained in time for effective management decision making.

BACKGROUND AND APPROACH

Over the past four years the author has studied the manpower vs time pattern of several hundred medium to large scale software development projects of different classes. These projects all exhibit a similar life cycle pattern of behavior – a rise in manpower, a peaking and a tailing off. Many of these projects (and all the large ones) follow a time pattern described by the life cycle curves of Norden (7,8) which are of the general Weibull class and more specifically the Rayleigh form,

$\dot{y} = K/t_d^2 \cdot t \cdot e^{-t^2/2t_d^2}$, where $\dot{y}$ is the manpower at any time t; K is the area under the curve and is the nominal life cycle effort in manyears; t_d is the time of peak manpower in years and corresponds very closely to the development time for the system.

Even though large systems seem to follow this general pattern, some small systems do not. They seem to have a more rectangular manpower pattern, The reason for this is that the applied manpower pattern is determined by management and by contractual agreements. Many small projects

are established as level-of-effort contracts – hence rectangular manloading. For large projects this is generally inadequate because managers have a poor intuitive feel for the resources to do the job. Accordingly, they tend to respond to the needs of the system reactively. This results in time lags and underapplication of effort at some instant in time, but the effect is a reasonably close approximation to Rayleigh manloading.

The author has shown in earlier works (5-6) that there is a Rayleigh law at work. It is the 1st subcycle of the overall development curve called the design and coding curve (detailed logic design and coding). This is also a manpower curve that is proportional to the analyst and programmer manpower – the direct productive manpower. This curve is denoted $\dot{y}_1$. Its form is

$\dot{y}_1 = K/t_d^2 \; t\, e^{-3t^2/t_d^2}$ (MY/YR) when related to the original definition of K and t_d for the overall burdened life cycle curve. When this curve is multiplied by the average productivity ($\overline{PR}$) for the project it yields the rate of code production.

$$\frac{dS_s}{dt} = \dot{S}_s = 2.49\, \overline{PR}\, \dot{y}_1, \text{ where the 2.49 is}$$

necessary to account for the definition of productivity as a burdened number (i.e., includes overhead and support activities). Now the time integral of the rate of code production yields the total number of source statements,

$$S_s = \int_0^\infty \frac{dS}{dt}\, dt = \overline{PR}\; 2.49 \int_0^\infty \dot{y}_1\, dt$$

$$S_s = \overline{PR} \,.\, 2.49.K/6.$$

The author has found that the $\overline{PR}$ is related to the Rayleigh parameters K and t_d in the following manner (6):

$\overline{PR} = C_n\, (K/t_d^2)^{-2/3}$ where the term K/t_d^2 has been defined as the system difficulty in terms of effort (K) and time (t_d) to produce it and C_n is a quantized constant defining a family of such curves. C_n is a channel capacity measure in the information theory sense, but in a more practical sense, it seems to be a measure of the state-of-technology being applied to a particular class of system.

Substituting for $\overline{PR}$, we obtain the software equation:

$$S_s = 2.49\, C_n (K/t_d^2)^{-2/3}\, K/6$$

$$S_s = \frac{2.49}{6}\, C_n\, K\, K^{-2/3}\, t_d^{4/3}$$

$$S_s = C_K\, K^{1/3}\, t_d^{4/3}, \text{ where } C_K \text{ has now subsumed } \frac{2.49}{6}\, C_n.$$

Having this expression which now relates the product in source statements to the Rayleigh manpower parameters (which are also the management parameters), we turn to a practical way in which

to estimate the size (S_s), effort (K) and development time (t_d) of a software project early in the requirements and specification phase of the project. This will let us answer the management questions necessary for effective investment decisions for the software project.

We will do this in the form of a case history for a project we will call SAVE. First, we will show a way to obtain a good estimate of the number of source statements. We'll plot the software equation and establish a feasible region for our development time parameters, we will impose a constraint relation involving K and (t_d). We will do a Monte Carlo simulation to generate variances for K and (t_d). With these numbers in hand, we can then do a trade-off analysis, pick a reasonable effort (cost) time combination and complete our translation into quantitative answers to the management questions. The answers we obtained will be close to optimal for the given constraint and, moreover, we will automatically have a sensitivity and risk profile.

INITIAL SIZING

Given the broad, preliminary design of SAVE consisting of the processing flow of the major functions and the estimates by the designers of the size range of the major functions, we can make a preliminary estimate of the development time, development effort and development cost to build the system.

The input data from the project team are in the form of size ranges for each major function. Three or four team members estimated the size of each function as follows:

- Smallest possible size (in source statements) – a
- Most likely size – m
- Largest possible size - b

These were averaged for each function and resulted in the first 3 columns of Table 1. This was in effect a Delphi polling of experts and their consensus. (Having done this with several groups of systems engineers, it is interesting to note that they are very comfortable with this procedure.)

Note that this results in a broad range of possible sizes for each function and that the distribution is skewed on the high side in most cases. This is typical of the Beta distribution, the characteristics of which are used in PERT estimating. We adopt the PERT technique to get an overall system size range and distribution.

1. An estimate of the expected value of a Beta distribution is:

$$E_1 = \frac{a + 4m + b}{6}$$

The overall expected value is just the sum of the individual expected values.

$$E = \sum_{i=1}^{N} E_i .$$

This is the sum of the fourth column of Table 1 (98475 S_s),

2. An estimate of the standard deviation of any distribution (including Beta) is the range within which 99% of the values are likely to occur divided by 6, i.e,

$$\sigma_i = |b-a| \ /6$$

The overall standard deviation is the square root of the sum of the squares of the individual standard deviations, i.e.,

$$\sigma_{tot} = \left(\sum_{i=1}^{N} \sigma_i^2 \right)^{1/2}$$

This results in a much smaller standard deviation than one would "guess" by just looking at the individual ranges: the reason is that some actuals will be lower than expected (E_i); others will be higher. The effects of these variations tend to cancel each other to some extent. This cancelling effect is best represented by the root of the sum of squares criterion.

The result is

$\hat{E}$ = 98475 source statements

$\hat{\sigma}_{tot}$ = ± 7081 source statements

and the 99% range is 77,000 – 120,000 S_s, or we are 99% sure that the ultimate size will be in this range if the input estimates do not change. Of course, if the input estimates change, we should redo our calculations and revise the results accordingly.

Major Function	Least a	Most Likely m	Most b	Expected E_i	Standard Deviation σ_i
Maintain	8675	13375	18625	13467	1658
Search	5377	3988	13125	9109	1258
Route	3160	3892	8800	4588	940
Status	850	1425	2925	1579	346
Browse	1875	4052	8250	4389	1063
Print	1437	2455	6125	2897	781
Cser Aids	6875	10625	16250	10938	1663
Incoming Msg	5830	3962	17750	9905	1987
Sys Monitor	9375	14625	28000	16979	3104
Sys Mgt	6300	13700	36250	16225	4992
Comm Proc	3875	3975	14625	9400	1458
				98475	7081

Table 1.

L. Putnam
QSM, INC.

DEVELOPMENT TIME-EFFORT DETERMINATION

Table 2 is a result of using the software equation which relates the product in source statements to the effort, development time and state-of-technology being applied to the project. The equation is derived partly from theory and partly from an empirical fit of a substantial body of productivity data. The form of the equation is:

$$S_s = C_k . K^{1/3} t_d^{4/3}$$

where S_s is the number of end product delivered source lines of code, an information measure.

C_k is a state-of-technology constant. For the environment anticipated for SAVE this constant is 10040. C_k can be determined by calibration against the software equation using data from projects developed by the same software house using similar technology and methods.

		t_d = 2 yrs	Fastest		Risk Biased	
	S_s	Dev Effort (MY)	t_d	Dev Effort (MY)	t_d + .4 yr	Dev Effort (MY)
-3σ	77000	11.28 ($.564M)	1.63	25.80 ($1.29M	2.03	10.71 ($.55M)
-1σ	91394	18.86 ($.943M)	1.75	32.16 ($1.61M)	2.15	14.12 ($.71M)
E	98475	23.59 ($1.18M)	1.81	35.40 ($1.77M)	2.21	15.91 ($.796M)
$+1\sigma$	105556	29.05 ($1.45M)	1.86	38.71 ($1.84M)	2.26	17.77 ($.89M)
$+3\sigma$	120000	42.69 ($2.135M)	1.97	45.55 ($2.28M)	2.37	21.77 ($1.09M)

Table 2. Assumptions: On-Line interactive development; top-down, structured programming; HOL; contemporary development environment. C_K = 10040, Standalone system – $|\nabla D|$ = 15.

K is the life cycle effort in man years. This is directly proportional to development effort

(Dev Effort = .4K) and cost ($\overline{\$/MY}$. K = $LC cost; $\overline{\$/MY}$. (.4K) = $ Dev).

t_d is the development time in years. This corresponds very closely to customer turnover.

Figure 1 shows a parametric graph of this equation.

Table 2 presents three scenarios for 5 different points in the size distribution curve. The expected case is given in the row labelled E. The column under t_d = 2 years gives a nominal development effort of 23.59 man years, $1.18M cost (@ $50,000/MY) to do 98475 source statements.

The fastest (or minimum) possible time for 98475 source statements is 1.81 years. The corresponding development effort is 35.4 MY, and cost of $1.77 million. The assumption here is that the system is a stand alone and the gradient condition of |70| = 15 cannot be exceeded.

The risk biased column is based on deliberately adding time (.4 of a year) to the minimum time to increase the probability of being able to deliver the product at the contract specified date. This biasing is to allow for external factors such as late delivery of a computer, an average number of requirements changes during development, etc. In the case of 98,475 source statements, this would be 1.81 +.4 = 2.21 years. The corresponding expected development effort is 15.91 MY; $.8 million cost. Note that development effort and cost go down as time to do the job is increased. This is Brooks' law at play. Conversely, there is no free lunch – if time is shortened the cost goes up, dramatically.

This can be illustrated by obtaining the trade off law from the software equation. Solve the software equation for K:

$$S_s = C_k K^{1/3} t_d^{4/3} = C_k (K t_d^4)^{1/3}$$

$$K t_d^4 = \left(\frac{S_s}{C_k}\right)^3$$

$$K = .4\left(\frac{Ss}{C_k}\right)^3 / t_d^4.$$

This is the trade-off law. In terms of development effort, E = .4K so

$$E = .4\left(\frac{S\, S_s}{C_k}\right)^3 / t_d^4 \text{ MY}$$

In our specific case

$$E = .4\left(\frac{98475}{10040}\right)^3 / t_d^4$$

and we can trade-off between 2 years (contract constraint, say) and 1.81 years – the minimum time for our gradient constraint.

PARAMETER DETERMINATION BY SIMULATION

While Table 2 gives a fairly broad range of solutions that answer many "what if" questions, it is an essentially deterministic solution; that is, it assumes we know the input information exactly. Of course, we don't.

A better solution, then, is one in which we treat the uncertainties in our input information in obtaining our solution. This is generally not feasible analytically, but is nicely handled by Monte Carlo simulation. In our case we do this by letting the input number of S_s vary randomly about the expected value (98,475) according to our computed standard deviation, $\sigma_{S_s} = 7081$, and letting the the stand-alone gradient ($|\nabla D| = 15$) vary within the statistical uncertainty of its measured (computed) value ($\sigma_D = 2$).

We then run the problem on the computer several thousand times with these random variations in parameters and generate the statistics of the variation in our answer. This is a much better measure of what is likely to happen as a result of the uncertainties in the problem.

The results of the simulation are given in the next table. Notice that the simulated estimated development effort is the same as the expected deterministic value and the development time is also the same. This is as it should be. The simulation produces the right expected values. The real value in the simulation is that it produces a measure of the variation in effort and in development time which we can used to construct risk profiles.

SAVE SIMULATION
INPUT: $S_s = 98475$, $\sigma_{S_s} = 7081$ $D = 15$, $\sigma_D = 2$ $C_k = 10040$, N = 2170 iterations
RESULTS: Expected development time = 1.81 yrs. σ, development time = ± .063 yrs. Expected development effort = 35.1 MY σ, development effort = ± 3.77 MY

Table 3.

MAJOR MILESTONE DETERMINATION

The results of the simulation determination of the development time are used to generate the major milestones of the project.

These milestones relate to the coupling of subcycles of the life cycle to the overall project curve (5). Examination of several hundred systems shows this coupling is very stable and predictable. The empirical milestones resulting from these earlier studies shows the following scaling.

Event	Milestone Fraction of Development Time, t_d
Critical Design Review	.43
Systems Integration Test	.67
Prototype Test	.80
Start Installation	.93
Full Operation Capability	1.0

Table 4 converts this to the appropriate descriptors and actual time schedule for this project.

SAVE MILESTONES (t_d = 1.81 years)		
Event	t/t_d	Time from start (months)
CDR	.43	9
Software S.I.T.	.67	15
Hardware S.I.T.	.80	17
Start Install.	.93	20
Start Accept. Test	1.0	22
Complete Accept. Test	1.14	25

Table 4.

RISK ANALYSIS

The results of the SAVE simulation for development time, development effort and development cost can be shown in the form of probability plots. Assuming a normal (gaussian) distribution, all that is necessary is an estimate of the expected value (plotted at 50% level) and the standard deviation (plotted offset from the expected value at the 16% probability level) to generate the line. Then one can determine the probability of any value of the quantity in question. For ease of presentation, the plots are summarized in Table 5.

% Probability that value will not be greater than	Dev Time (t_d) years	Dev Effort (E) manyears	Dev Cost PH II Millions
1	1.55	25	1.25
10	1.73	30	1.50
20	1.76	32	1.60
50	1.81	35.1	1.75
80	1.86	38.5	1.93
90	1.90	40	2.04
99	1.97	45	2.25

Table 5.

The result for the development time is extremely important from a conceptual point of view. The small standard deviation is both a curse and a blessing. It says we can determine the development time very accurately (σ_{t_d}/t_d = 3.5%) but at the same time it tells us we have little latitude in adjusting the development time to meet contractual requirements.

For example, σ_{t_d} = .063 years is .063 (52) = ± 3.28 weeks; $3\sigma_{t_d}$ = 3 (3.28) = ± 9.83 weeks;

= ± 9.83

= ± 10 weeks

So, if we add 30 to t_d we will be 99% sure that t_d will not exceed the actual value from random causes. This does not mean that requirements changes or late delivery of a computer will still permit the software to come in at ± 10 weeks of the expected time. These are external factors that will change t_d and must be specifically accounted for.

This is the curse. The system is very sensitive to external perturbations and these will generally cause development time increments greater than 2 or 3 σ_{t_d} (a 90 day delay in test bed computer delivery, say).

But, knowing this great time sensitivity, management can use it effectively in planning and contracting so that risk is always acceptable. The major point is: time is not a free good. Development time cannot be specified by management.

MANPOWER AND CASH FLOW PATTERN

Now that we have the parameters for development effort and development time we can generate the manloading and cash flow pattern for the software development period (and even the life cycle, if we choose). The Rayleigh/Norden equation gives the instantaneous manpower.

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

for the software development effort (Phase III). The cash flow is just the average dollar cost/MY times $\dot{y}$.

$$\text{Cash Flow Phase II} = \overline{\text{S/MY}} \cdot \dot{y} \text{ \$/YR}$$

Table 6 combines the software development effort (Phase II) with the initial design and system specification (Phase II) overlap and the hardware integration and test effort. The column labelled total adds the separate efforts together at each time period to show the total people on board. The cash flow rate is the annualized spending rate at that instant in time (assuming an average burdened cost/MY of $50,000). The last column gives the cumulative cost at each two month interval.

(Mos.)	PH II	PH I	HDWRE	TOTAL	CASH FLOW RATE ($ MIL/YR)	SUM COST ($ MIL.)
	. . . People					
0	0	0	10		.50	– –
2	5	0	13		.65	.096
4	9	0	15		.80	.217
6	13	1	18		.90	.358
8	17	1	19		.95	.513
10	21	2	23		1.15	.888
12	24	3	28		1.25	.896
14	25	3	29		1.45	1.383
16	28	4	32		1.60	1.654
18	29	4	33		1.65	1.933
20	30	4	34		1.70	2.225
22	30	6	36		1.80	2.405
24	20	4	24		1.20	

Table 6.

Figure 2 shows the time-phased manloading of the Phase II part of the project as laid out in Table 6.

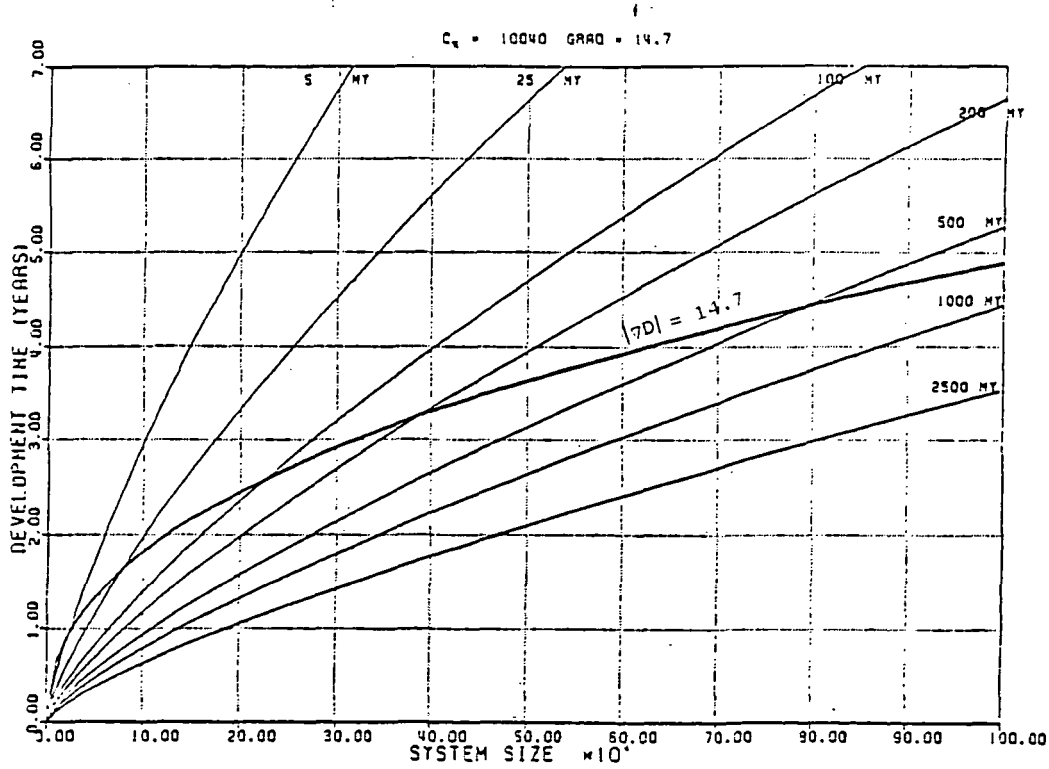


Figure 1. Size - Effort - Time Trade-Off Chart

L. Putnam
QSM, INC.

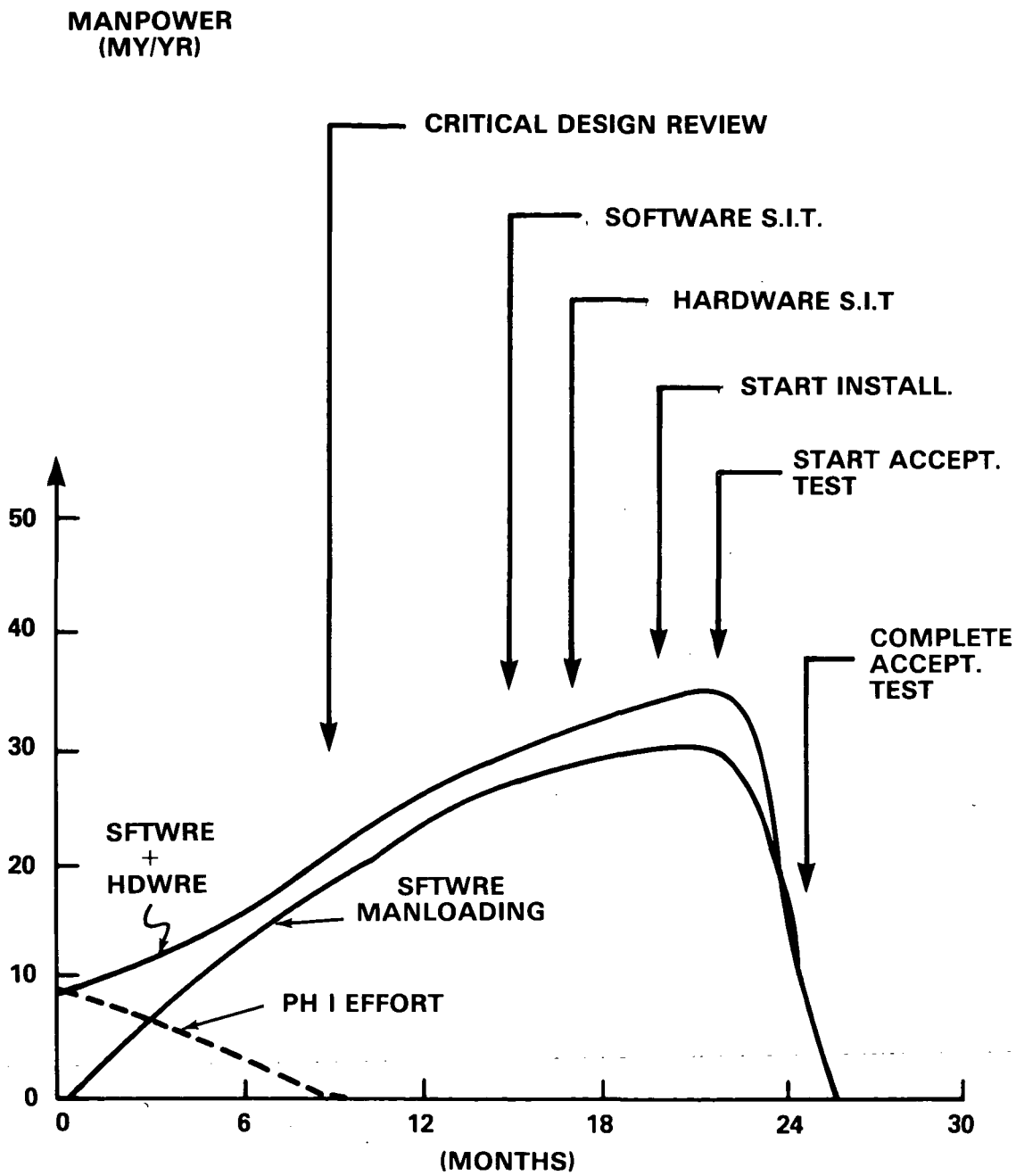


Figure 2

L. Putnam
QSM, INC.

CONCLUSION

We have shown that the management questions posed at the beginning can be answered quantitatively to acceptable engineering accuracy for a software project during the specification preparation phase. We need only know the state-of-technology we are going to apply to the development, estimate the number of lines of code using the PERT techniques, and use the software equation with a constraint relationship to solve for the management parameters (K,t_d) of the Rayleigh/Norden equation. Simulation provides suitable statistics for risk estimation.

REFERENCES

1. Brooks, F.P. Jr., The Mythical Man-Month, Addison-Wesley Publishing Co., Reading, MA. 1975.

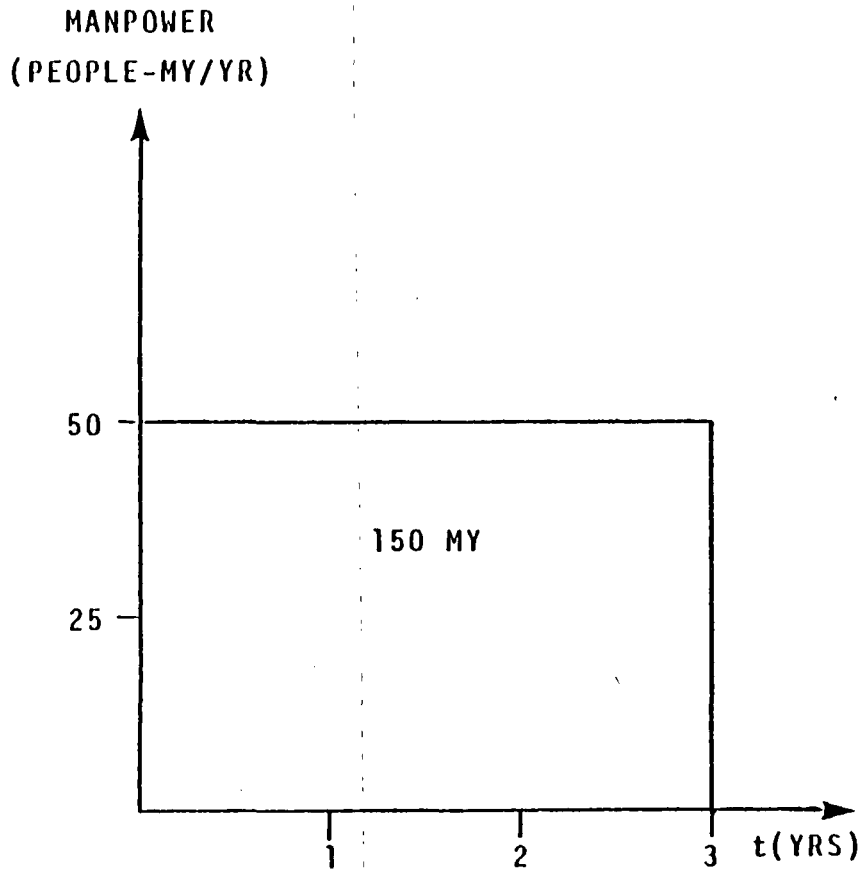
2. Morin, Lois H., Estimation of Resources for Computer Programming Projects, MS Thesis, Univ. of North Carolina, Chapel Hill, N.C., 1973.

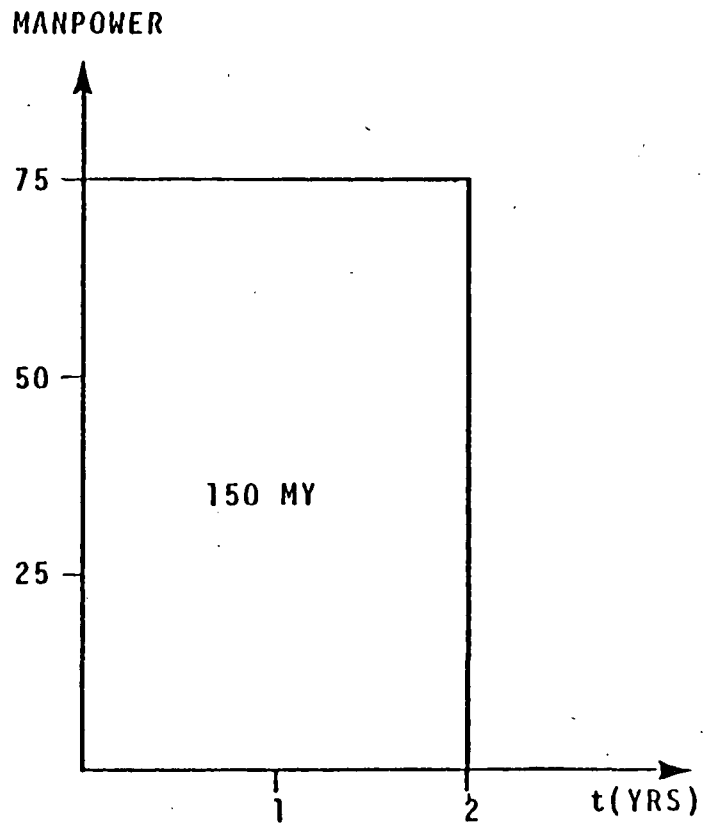
3. Norden, Peter V., "Useful Tools for Project Management." Management of Production, M.K. Starr (Editor), Penguin Books, Inc., Baltimore, Md., 1970, pp. 71-101.

4. Norden, Peter V., Project Life Cycle Modelling: Background and Application Of The Life Cycle Curves, Papers from the Software Life Cycle Management Workshop, Airlie, Va., Aug. 1977, sponsored by US Army Computer Systems Command.

5. Putnam, Lawrence H., and Wolverton, Ray W., Quantitative Management: Software Costing Estimating, A Tutorial for COMPSAC '77, The IEEE Computer Society's First International Computer Software and Applications Conference, Chicago, Ill., 8-10 Nov. 1977.

6. Putnam, Lawrence H., "A General Empirical Solution to the Macro Software Sizing and Estimating Problem," to appear in IEEE Transactions on Software Engineering, Summer, 1978.





TYPICAL SOFTWARE ASSUMPTIONS:

- PRODUCTIVITY $\left(\frac{S_s}{MY}\right)$ IS CONSTANT AND CAN BE DETERMINED BY MANAGEMENT
- PRODUCT (S_s) IS DIRECTLY PROPORTIONAL TO EFFORT (MY)

WHAT DO WE WANT TO KNOW ABOUT QUANTITATIVE SOFTWARE MANAGEMENT?

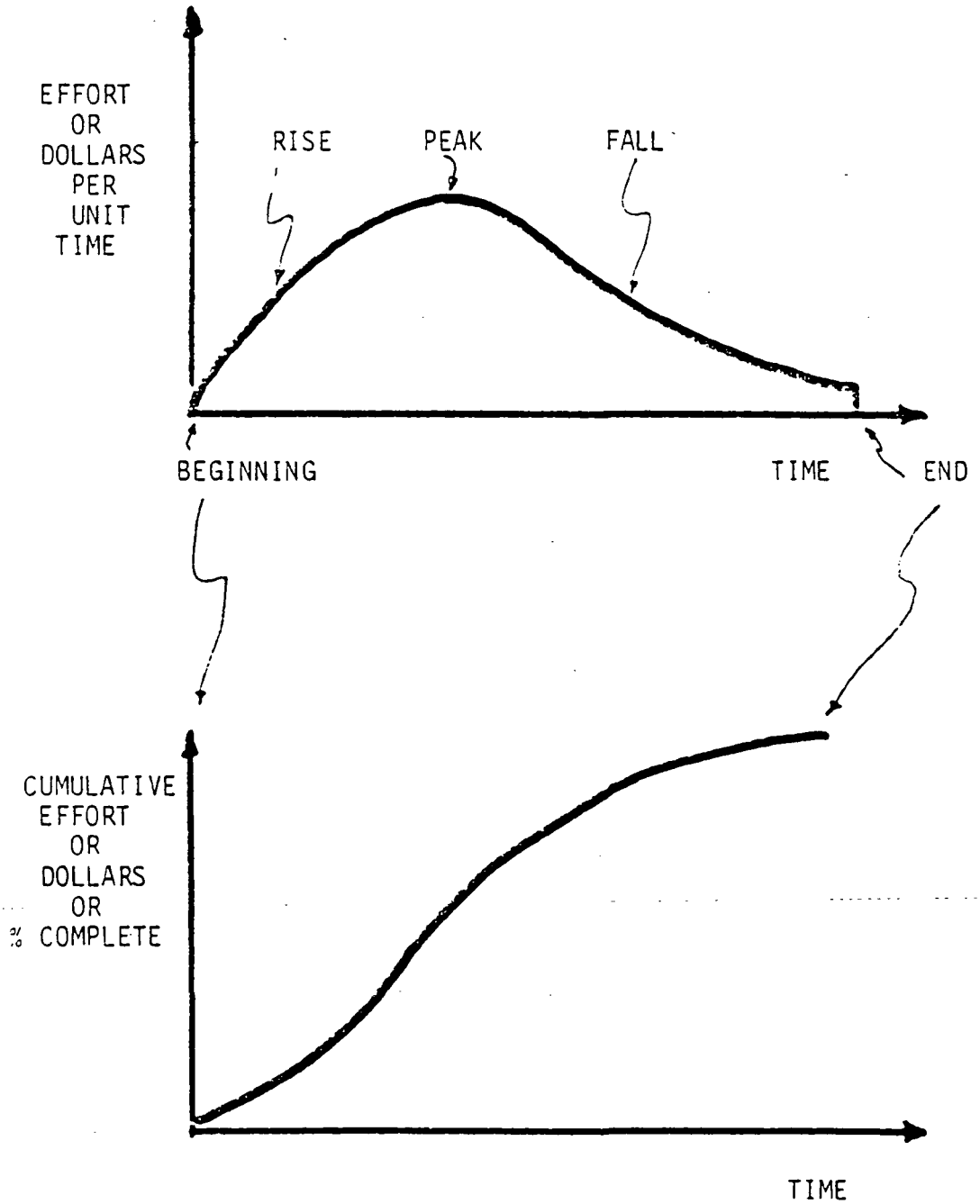
GEMENT METERS

- HOW MANY PEOPLE
- HOW LONG
- HOW MANY DOLLARS
- MANPOWER (MANLOADING AT ANY POINT IN TIME)
- CASH FLOW (SPENDING RATE)
- RISK OR UNCERTAINTY ASSOCIATED WITH EACH OF THESE
- TRADE-OFFS

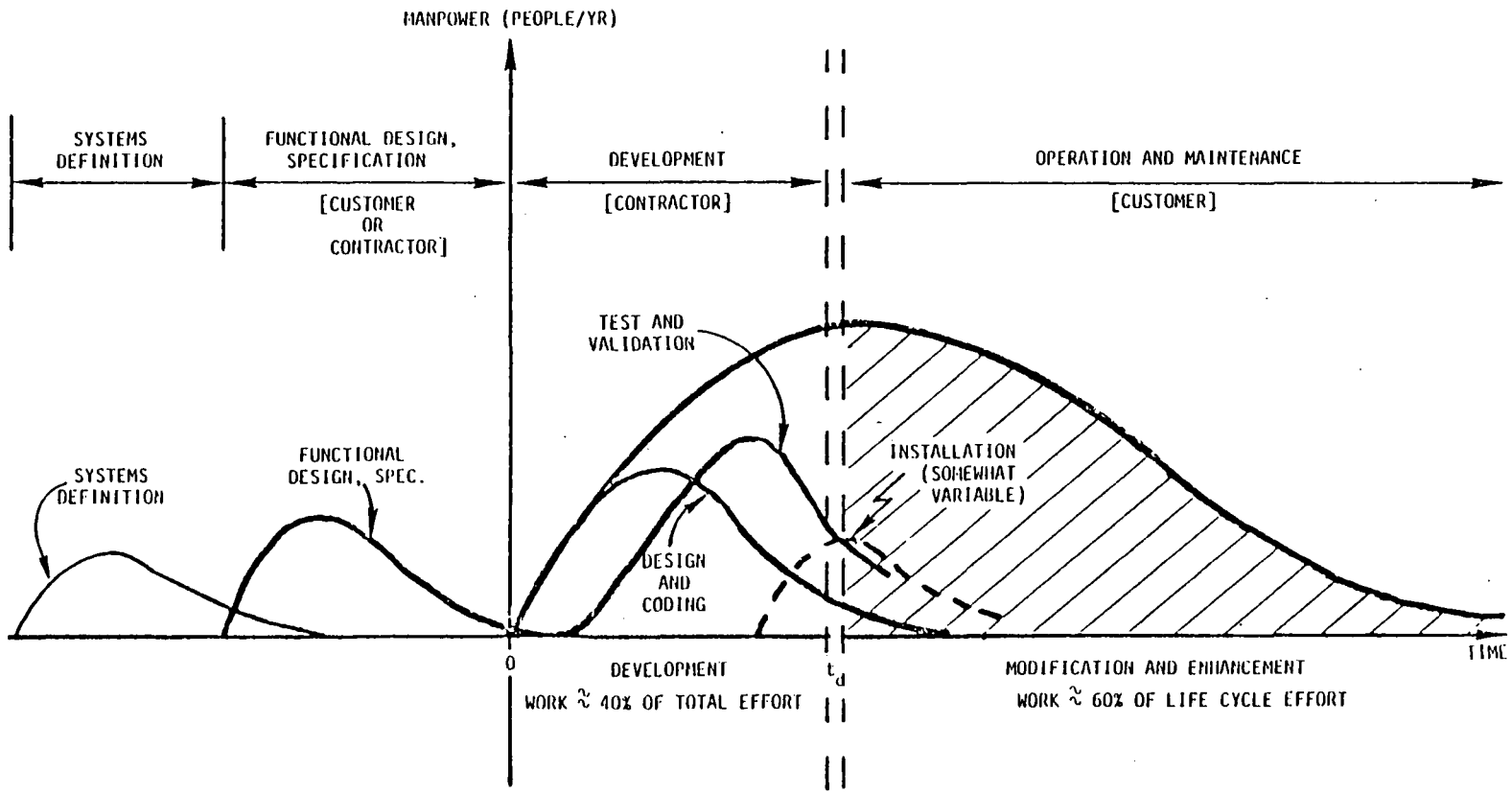
WHAT DO WE NEED?

- A SMALL SET OF EARLY (PERHAPS GROSS) SYSTEM CHARACTERISTICS THAT
- MAP INTO THE MANAGEMENT PARAMETERS
- A MEANS TO UPDATE (AT ANY POINT IN LIFE-CYCLE) AND CONTROL.

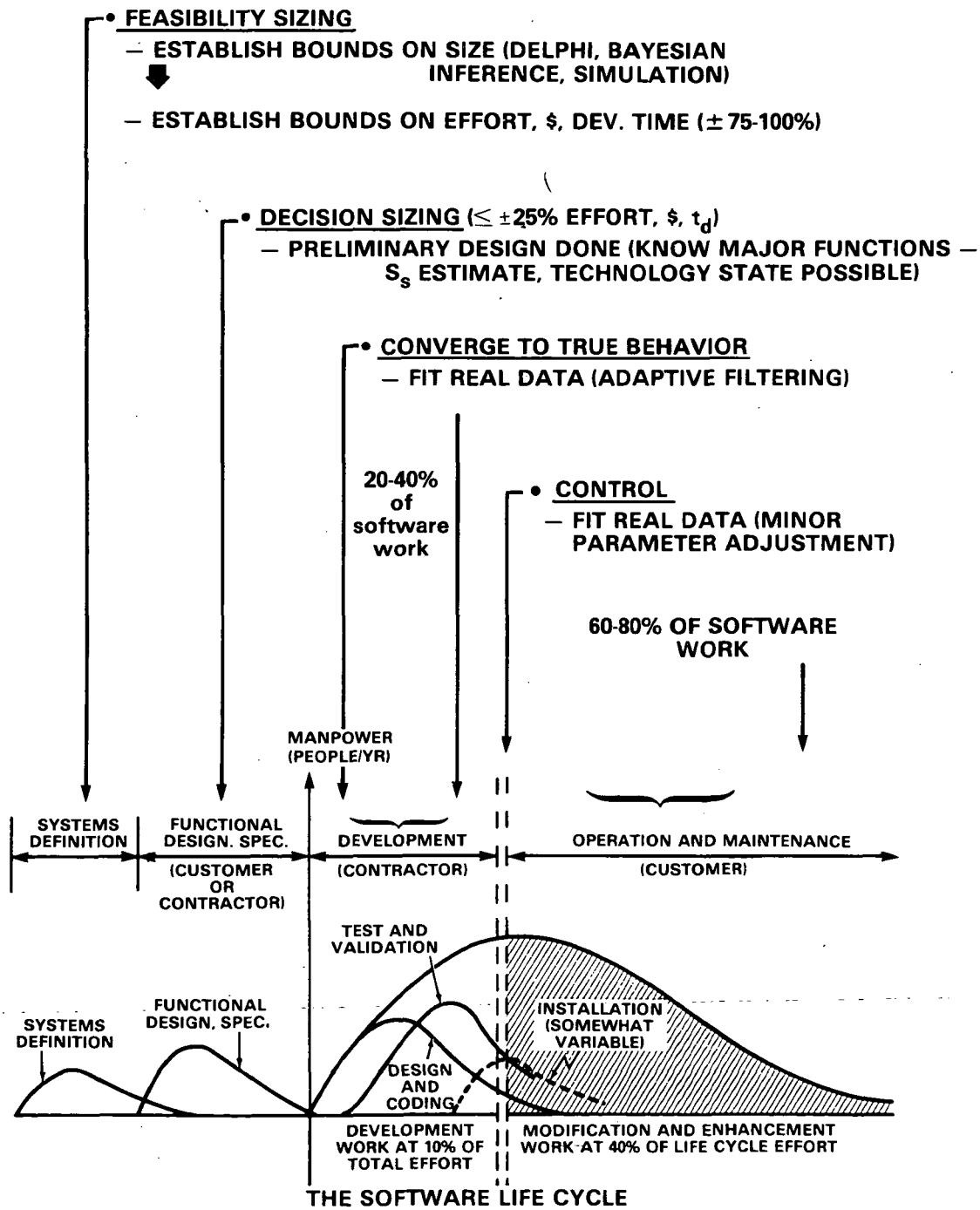
WHAT IS A LIFE CYCLE?



THE SOFTWARE LIFE CYCLE



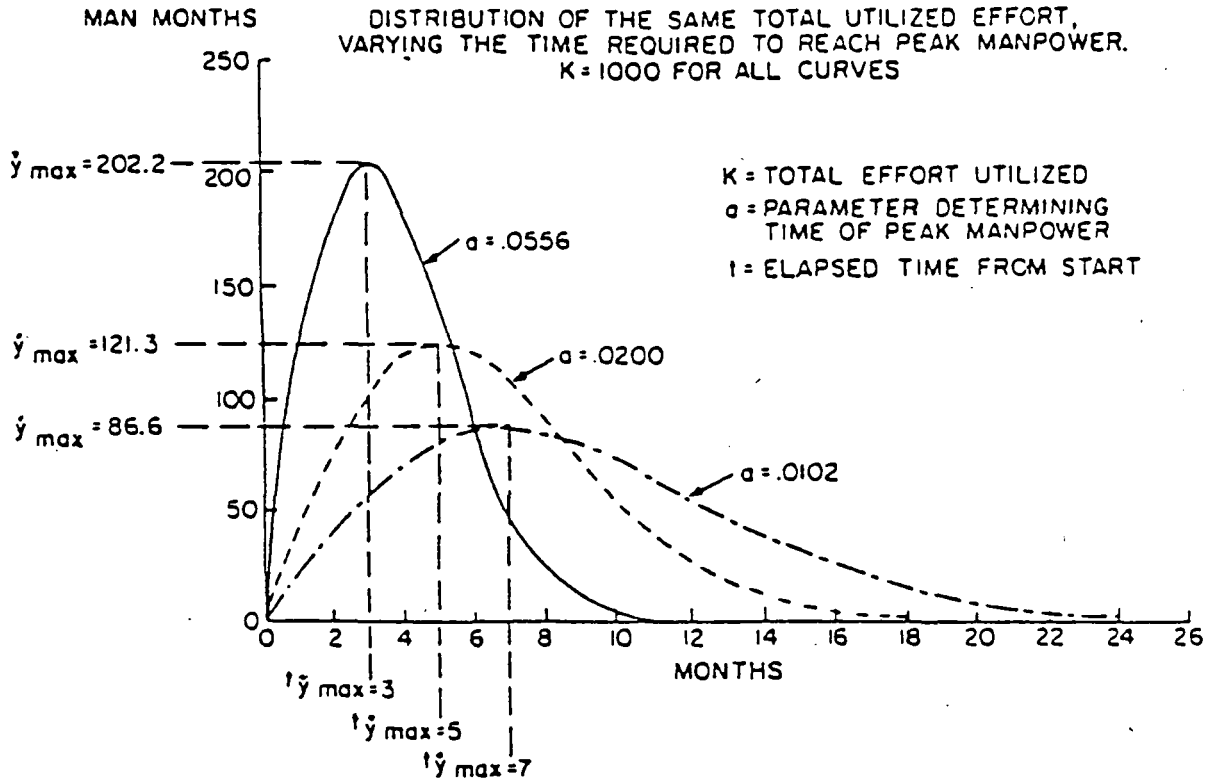
APPLICATION SOFTWARE: SOLVING THE SIZING-ESTIMATING PROBLEM

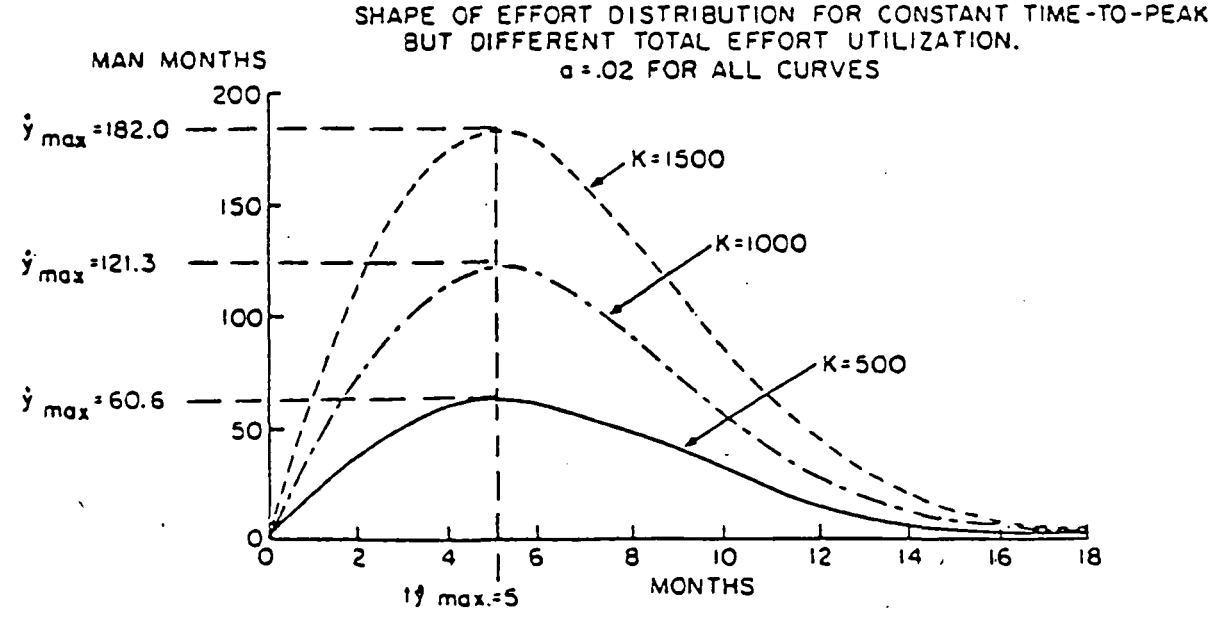


L. Putnam
QSM, INC.

MANPOWER UTILIZATION CURVE

$$\dot{y} = 2Kate - at^2$$



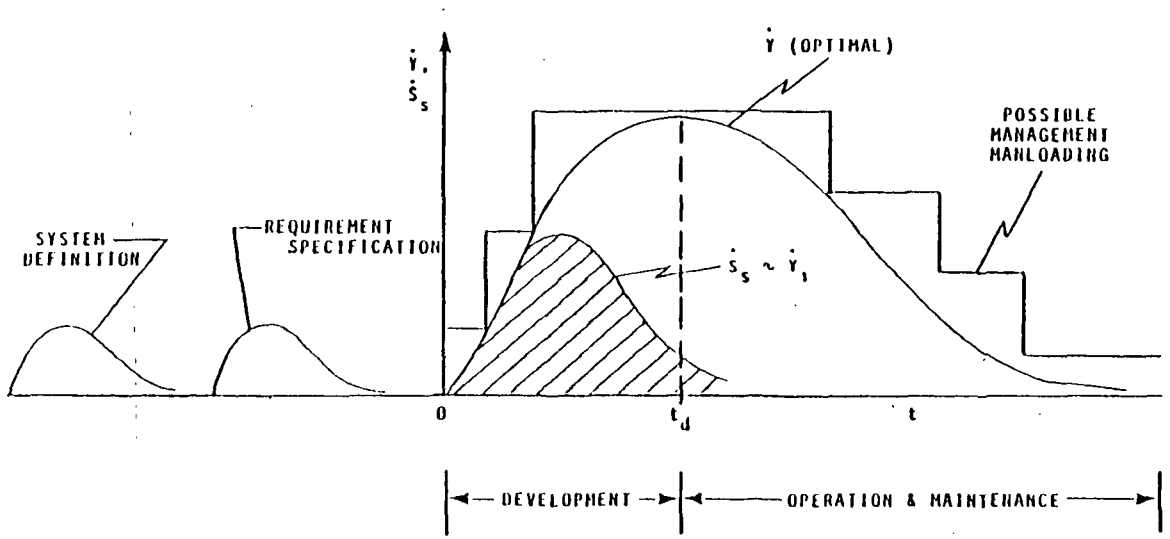


L. Putnam
QSM, INC.

WHY RAYLEIGH/NORDEN?

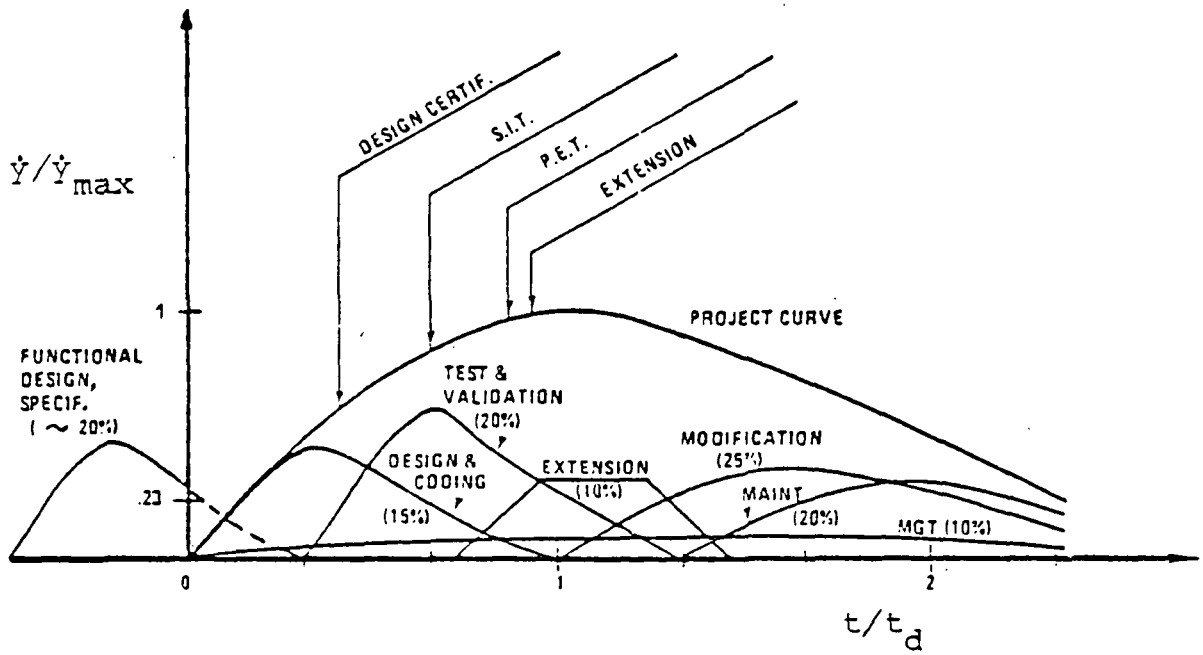
EVEN THOUGH OVERALL MANPOWER PATTERN OFTEN RESEMBLES RAYLEIGH/NORDEN FORM --

- IT DOES NOT HAVE TO -- MANAGEMENT DECIDES THAT.
- BUT RAYLEIGH PATTERN IS OPTIMAL -- SO IT IS THE BEST MANAGEMENT CHOICE.
- WHAT DOES HAVE TO BE RAYLEIGH IS THE CODE PRODUCTION RATE ($\dot{S}_s$).



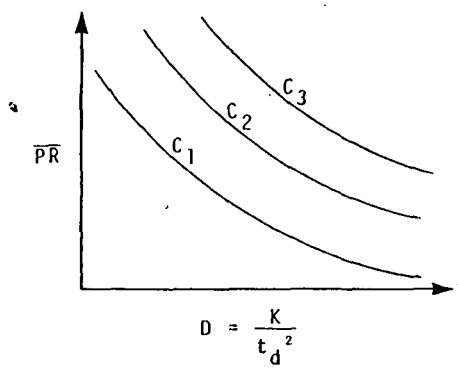
- THE SOFTWARE EQUATION IS BASED ON THE DESIGN AND CODING CURVE ($\dot{Y}_1 \sim \dot{S}_s$).
  IT MAPS INTO THE OVERALL MANLOADING CURVE FOR LARGE PROJECTS ($\dot{Y}_1$, $\dot{S}_s \rightarrow K$, t_d, t).

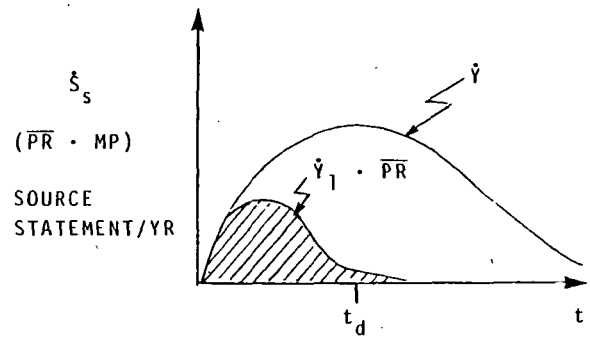
L. Putnam
QSM, INC.
Page 19 of 40



L. Putnam
QSM, INC.

THE SOFTWARE EQUATION



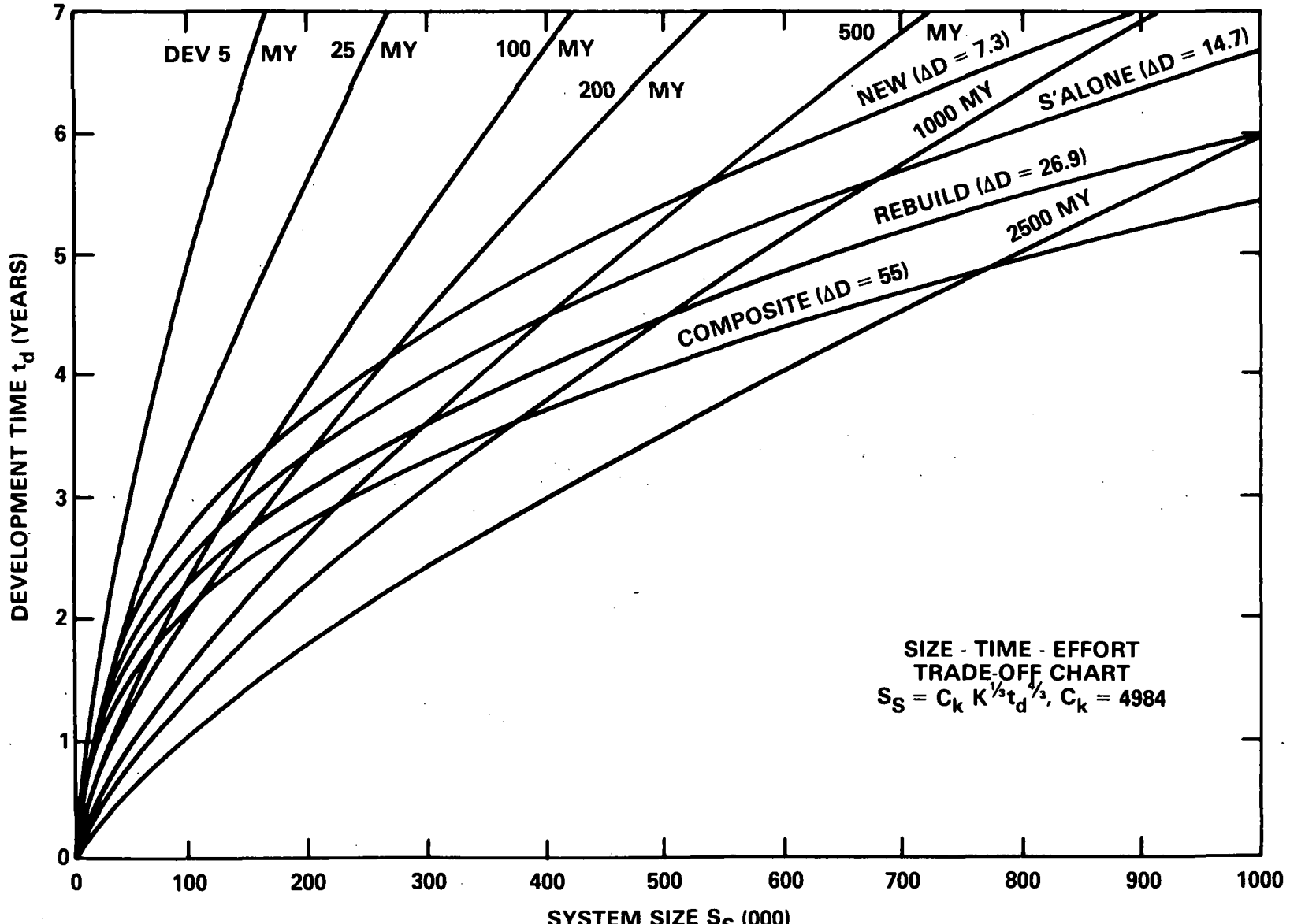


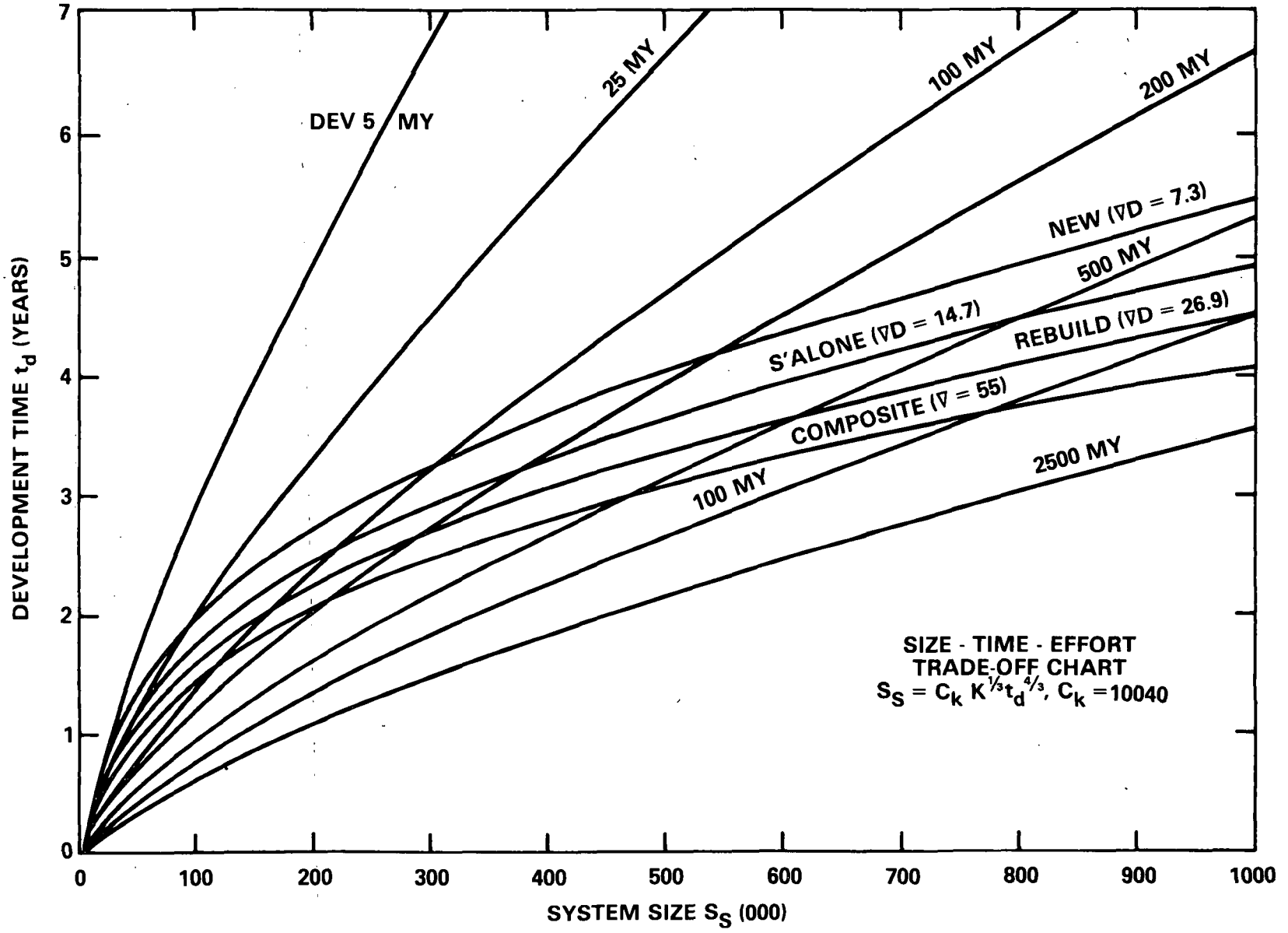
$$S_s = \int_0^\infty \dot{S}_s \cdot dt = \int_0^\infty \overline{PR} \cdot \dot{y}_1 \cdot dt = \overline{PR} \cdot \int_0^\infty \frac{K}{t_d^2} \cdot t \cdot \exp\left(\frac{-3t^2}{t_d^2}\right) \cdot dt$$

$$S_s = C_K \cdot \frac{1}{K^3} \cdot t_d^{\frac{4}{3}}$$

OUTPUT — STATE OF TECHNOLOGY PARAMETER — INPUT

- S_s IS THE PRODUCT (OUTPUT) - DEPENDENT ON SYSTEM CHARACTERISTICS
- C_K IS THE CHANNEL CAPACITY CONSTANT - QUANTIZED AND TECHNOLOGY DEPENDENT (MACHINE THRU-PUT, TOOLS, LANGUAGE)
- K, t_d ARE THE MANAGEMENT PARAMETERS (INPUT) - K, t_d MAY BE TRADED-OFF TO MATCH SYSTEM AND TECHNOLOGY CHARACTERISTICS AND POSSIBLY CONTRACTUAL CONSTRAINTS


DEV 5 MY
25 MY
100 MY
200 MY
500 MY
1000 MY
2500 MY
NEW (ΔD = 7.3)
S'ALONE (ΔD = 14.7)
REBUILD (ΔD = 26.9)
COMPOSITE (ΔD = 55)
SIZE - TIME - EFFORT
TRADE-OFF CHART
$S_S = C_k K^{1/3} t_d^{4/3}$, $C_k = 4984$
DEVELOPMENT TIME t_d (YEARS)
SYSTEM SIZE S_S (000)
0
1
2
3
4
5
6
7
0
100
200
300
400
500
600
700
800
900
1000



L. Putnam
QSM, INC.

TRADEOFF LAW

(COROLLARY TO BROOKS' LAW)

$K \sim$	$E = \frac{C_1}{t_d^4}$

NUMBER OF SOURCE STATEMENTS IS FIXED

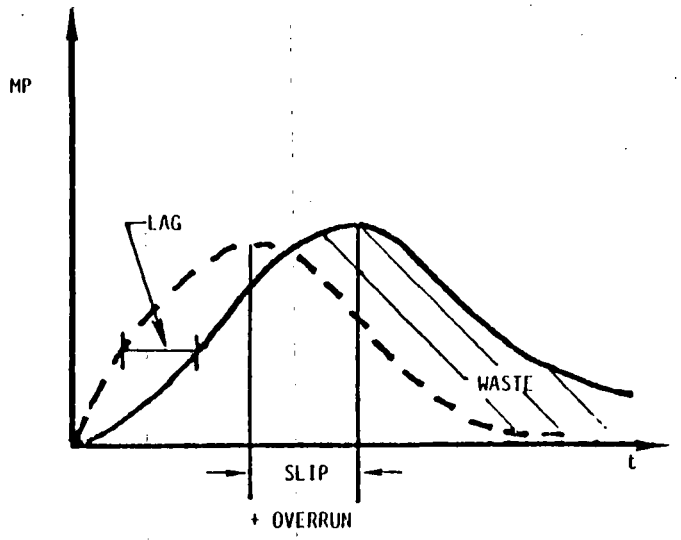
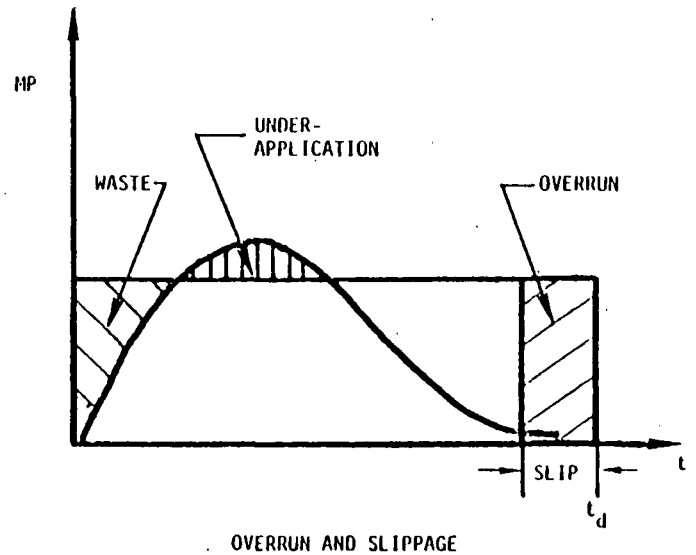
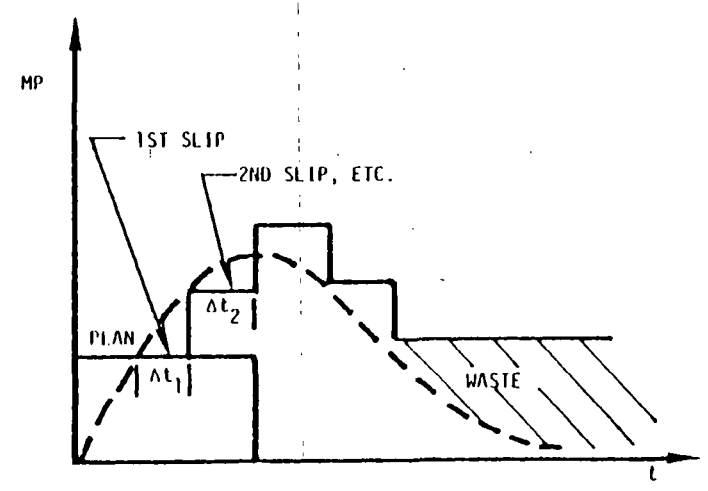
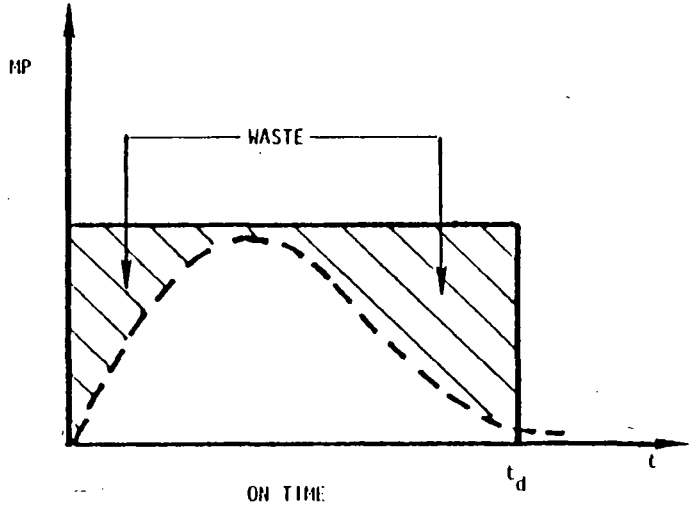
$$DEV\ \$ = \frac{C_2}{t_d^4}$$

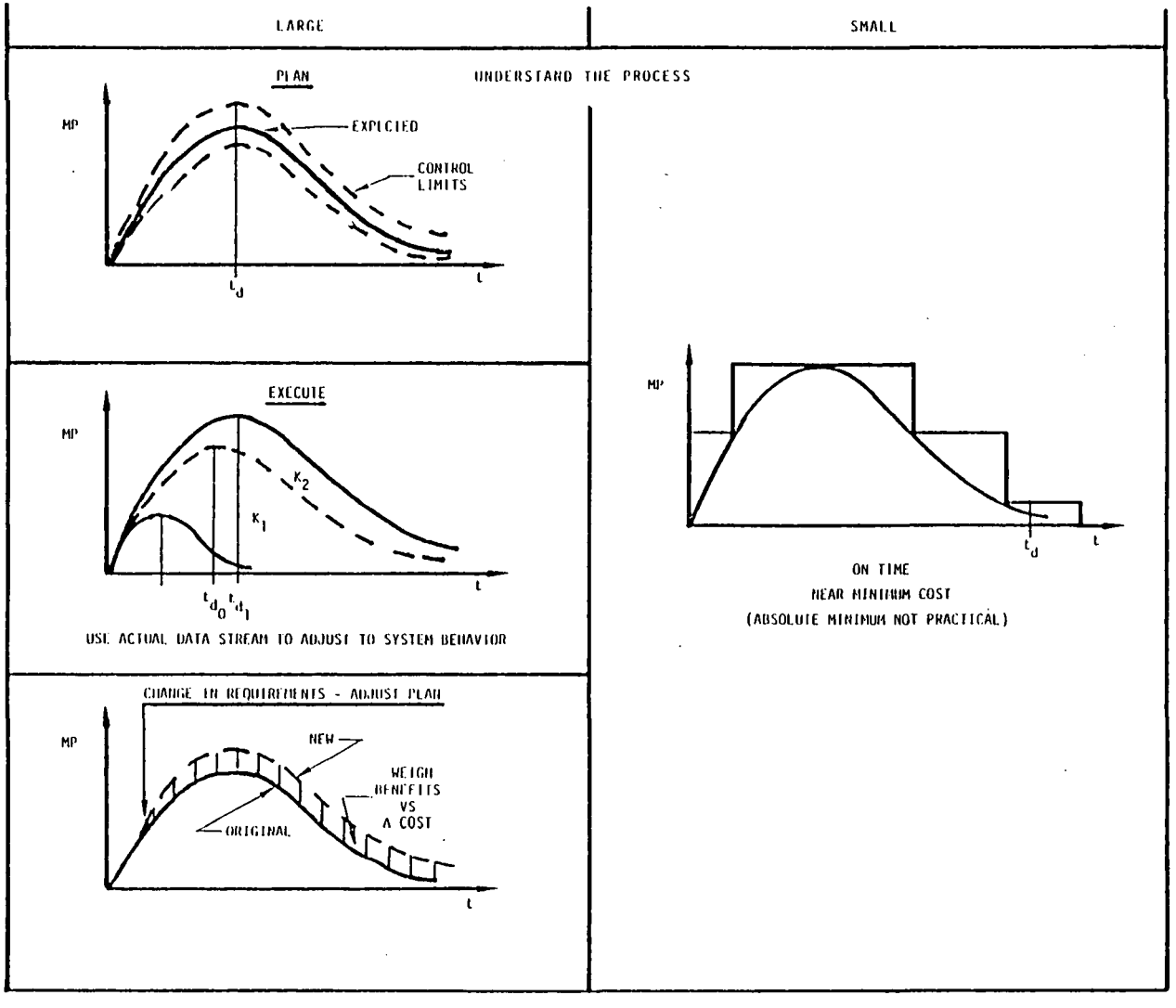
C_2 NOW SUBSUMES THE AVERAGE $ COST/MY

- SUBJECT TO THE GRADIENT CONSTRAINT

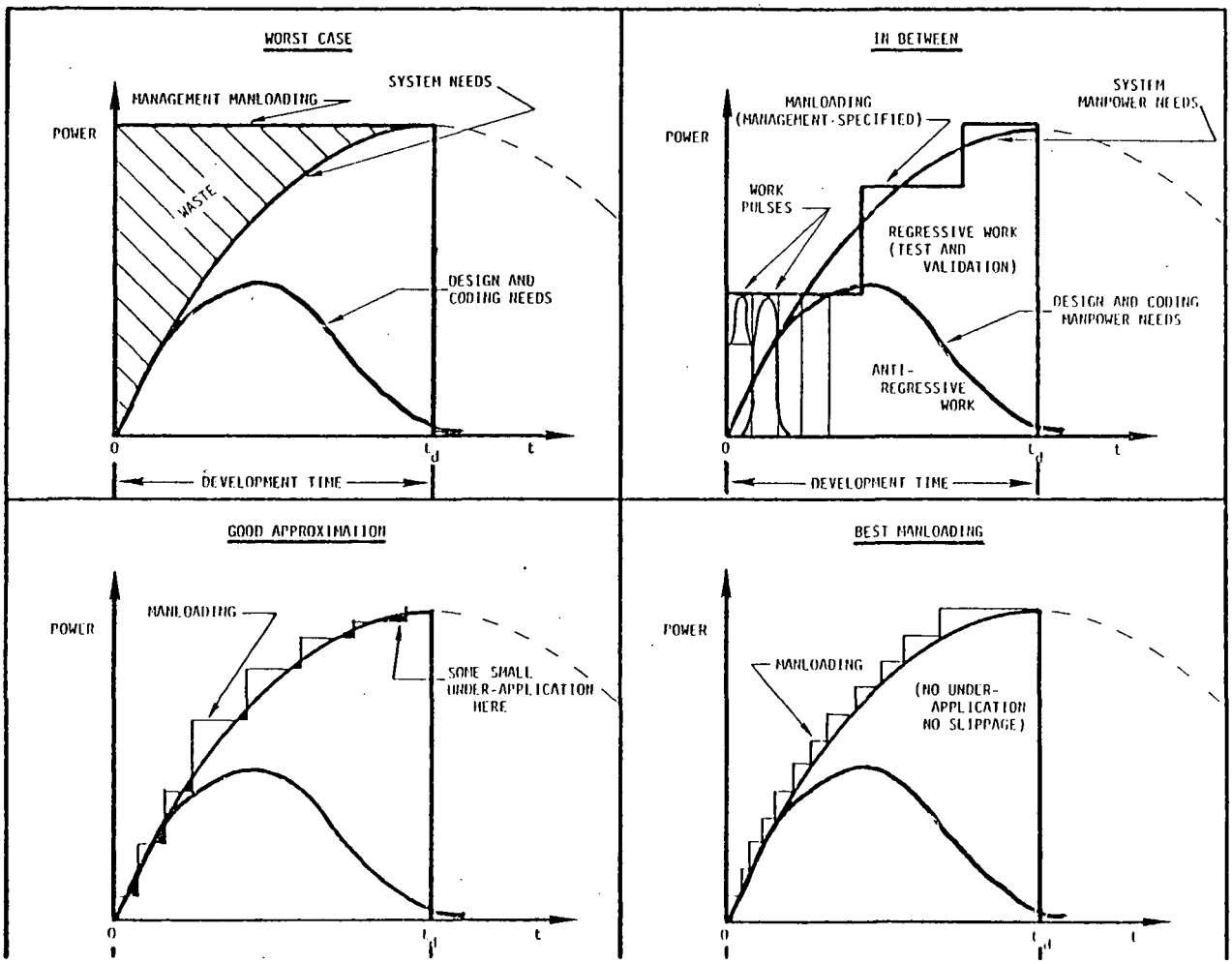
  (CANNOT EXCEED CAPABILITY OF ORGANIZATION AND ITS TECHNOLOGY)

EVOLUTION OF MANPOWER APPLICATION
1 of 2

LARGE	SMALL
NAIVE BEGINNER	
 + OVERRUN RESPONDS LATE TO SYSTEM DEMANDS	 OVERRUN AND SLIPPAGE
PLAN THE JOB	
 MULTIPLE SLIPS, ADD MANPOWER WHEN SLIPS OCCUR	 ON TIME COSTS TOO MUCH

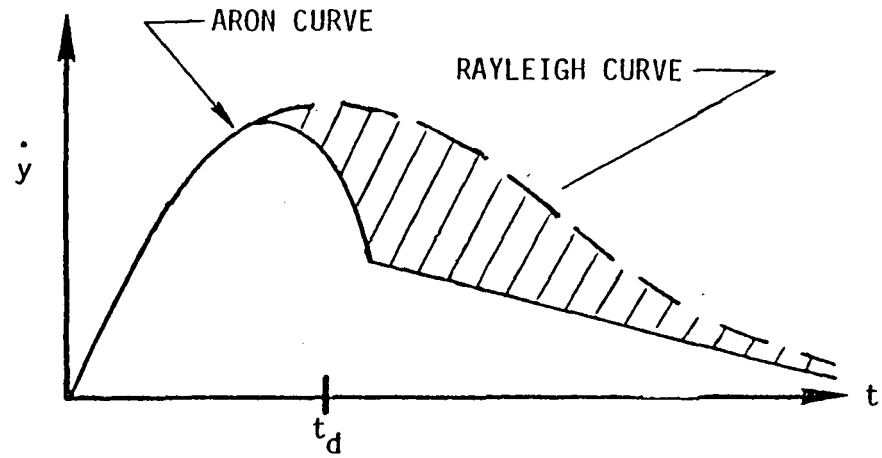

LARGE
SMALL
PLAN
UNDERSTAND THE PROCESS
MP
EXPECTED
CONTROL LIMITS
t_d
t
EXECUTE
MP
K_2
K_1
t_{d_0} t_{d_1}
t
USE ACTUAL DATA STREAM TO ADJUST TO SYSTEM BEHAVIOR
CHANGE IN REQUIREMENTS - ADJUST PLAN
MP
NEW
WEIGH BENEFITS VS A COST
ORIGINAL
t
MP
t_d
t
ON TIME
NEAR MINIMUM COST
(ABSOLUTE MINIMUM NOT PRACTICAL)

VARIOUS MANLOADING PATTERNS



L. Putnam
QSM, INC.

ARON (IBM) LIFE CYCLE CURVE IS NOT INCONSISTENT



CROSS-HATCHED AREA REPRESENTS

- EXTENSION
- ENHANCEMENTS

THIS WORK IS NORMALLY DONE BY CUSTOMER OR CUSTOMER REPRESENTATIVES (NOT SEEN BY SOFTWARE HOUSE).

THE INPUTS NECESSARY TO FORCAST SOFTWARE COSTS

- NO. OF FILES SYSTEM WILL HAVE
- NO. OF OUTPUT FORMATS SYSTEM WILL HAVE
- NO. OF APPLICATION SUBPROGRAMS SYSTEM WILL HAVE
- NO. OF SOURCE STATEMENTS SYSTEM WILL HAVE
- AVERAGE NO. OF SOURCE STATEMENTS PER SUBPROGRAM
- AVERAGE COST PER MAN YEAR OF EFFORT
- AVAILABILITY OF COMPUTER TEST TIME THROUGHOUT DEVELOPMENT – HOURS/MONTH
- DEVELOPMENT ENVIRONMENT
  - INTERACTIVE?
  - BATCH?
  - DEDICATED TO DEVELOPMENT, OR PRODUCTION WORK ALSO BEING DONE?
- MODERN SOFTWARE ENGINEERING TOOLS TO BE USED
- TYPE SYSTEM (BUSINESS, C&C, SCIENTIFIC, REAL-TIME, ETC.)
- TECHNOLOGY CONSTANT CALIBRATION DATA (DEV EFFORT, DEV TIME, SIZE IN SOURCE STATEMENTS (LESS COMMENTS) FOR ONE OR MORE PREVIOUS SIMILIAR SYSTEMS THAT USED A SIMILAR DEVELOPMENT ENVIRONMENT AND TOOLS)

DATA TO CAPTURE DURING DEVELOPMENT TIME

- RATE OF CODE PRODUCTION (S_s/YR)
- MANPOWER (RATE) DEVOTED TO DESIGN AND CODING ($\dot{y}_1$)
- CUMULATIVE CODE PRODUCTION AT TIME, t (S_s (t))
- CUMULATIVE PEOPLE ASSIGNED TO DESIGN AND CODING AT TIME T (y_1 (t))
- TOTAL CUMULATIVE PEOPLE (INCLUDING ALL INDIRECT EFFORT AND OVERHEAD) AT TIME t (y(t))
- ACTUAL TIMES WHEN CRITICAL EVENTS START
  - CRITICAL DESIGN REVIEW
  - SYSTEM INTEGRATION TEST
  - PROTOTYPE TEST (1ST ON-SITE, FULL SCALE TEST)
  - INITIAL OPERATIONAL CAPABILITY
  - CUSTOMER TURNOVER (IF DIFFERENT FROM I.O.C.)
- COMPUTER TEST TIME USED PER MONTH ($CH/_{MO}$)
- CUMULATIVE COMPUTER TEST TIME USED AT TIME t, (CH)

REASONS WHY SOFTWARE DETERIORATES

- RECOGNITION OF ORIGINAL DESIGN FAULTS.
- DISCOVERY OF BUGS/ERRORS.
- EVOLUTION OF A LEARNING USER WHO DEVELOPS HIS UNDERSTANDING OF THE "REAL" PROBLEM AND UPGRADES HIS REQUIREMENTS BASED ON OPERATIONAL EXPERIENCE.
- CHANGES IN THE APPLICATION ENVIRONMENT AS A RESULT OF BUSINESS, ACCOUNTING AND GOVERNMENT REQUIREMENTS.
- CHANGES IN COMPUTER TECHNOLOGY – BOTH SYSTEMS SOFTWARE AND HARDWARE.

Werner L. Frank
in COMPUTERWORLD

∴ "MAINTENANCE" (ENHANCEMENTS, MODIFICATIONS, ERROR FIXING)

DATA REQUIRED DURING OPERATIONS AND MAINTENANCE PHASE

- MANPOWER DURING DEVELOPMENT (MY/yr)
- TOTAL DEVELOPMENT EFFORT (MY)
- DEVELOPMENT TIME (FROM START OF DESIGN AND CODING TO CUSTOMER TURNOVER) (t_d)
- ELAPSED TIME FROM START TO START OF CRITICAL MILESTONES
- ACTUAL MANPOWER (CONTINUOUSLY AS A FUNCTION OF TIME) ($\dot{y}_{act}$)
- MAINTENANCE DATA
  - NO. ENHANCEMENTS STARTED/MO.
  - NO. EMERGENCY FIXES STARTED/MO.
  - NO. VALID ERRORS FOUND/MO.
  - NO. ENHANCEMENTS/FIXES DEFFERED/MO.
  - NO. MODULES CHANGED/MO.
- SIZE OF SYSTEM – SOURCE STATEMENTS (CONTINUOUSLY) (S_s)
- CUMULATIVE NO. MODULES/SUBPROGRAMS CHANGED SINCE TURNOVER (t_d)

SOFTWARE AXIOMS FOR PROJECT MANAGERS

- SOFTWARE DEVELOPMENT HAS ITS OWN CHARACTERISTIC BEHAVIOR
- SOFTWARE DEVELOPMENT IS DYNAMIC – NOT STATIC
- PRODUCTIVITY AND CODING RATES ARE CONTINUOUSLY VARYING – NOT CONSTANT
- PRODUCTIVITY RATES ARE A FUNCTION OF THE SYSTEM DIFFICULTY – MANAGEMENT CANNOT ARBITRARILY INCREASE PRODUCTIVITY. MANAGEMENT CAN FAVORABLY INFLUENCE THIS BY PROVIDING SUFFICIENT TIME.
- BROOKS' LAW GOVERNS – TIME AND MANPOWER ARE NOT FREELY INTERCHANGEABLE. (SHORTENING THE "NATURAL" DEVELOPMENT TIME OF A SYSTEM IS VERY COSTLY – AND MAY BE IMPOSSIBLE)
- THERE IS A SOFTWARE LAW THAT MUST BE OBEYED – OTHERWISE SLIPPAGE AND OVERRUN ARE INEVITABLE.
- KEEP A RECORD OF WHAT HAPPENED, WHEN AND HOW MUCH – IT WILL HELP NEXT TIME.

MENU
(WHAT WE CAN DO NOW)

- PARAMETER ESTIMATION
  - LIFE CYCLE SIZE (COST) (K)
  - DEVELOPMENT TIME (t_d)
- MANPOWER VS. TIME
- CASH FLOW VS. TIME
- COMPUTER TIME VS. TIME
- RISK ANALYSIS
  - COST
  - MANPOWER
  - TIME
- UPDATING ESTIMATES FROM ACTUAL DATA (BOX'S METHOD)
- DYNAMIC MODELING OF CHANGES TO RQMTS, SPECS
- SIMULATION OF MANPOWER, CASH FLOW
- LIFE CYCLE COST/BENEFIT ANALYSIS
- AGGREGATION OF SYSTEMS TO CONTROL TOTAL EFFORT OF SOFTWARE HOUSE
- FORECAST INTERNAL MANPOWER GENERATION RATE OF SOFTWARE HOUSE DOING MOSTLY MAINTENANCE WORK

WHAT DO WE NEED TO
ANSWER THE MANAGEMENT QUESTIONS?

ESTIMATES OF:

- NUMBER OF SOURCE STATEMENTS
- TECHNOLOGY CONSTANT
- ONE OR MORE CONSTRAINTS:
  - MANPOWER
  - MAXIMUM TIME
  - MAXIMUM COST
  - (MAXIMUM DIFFICULTY)
  - (MAXIMUM DIFFICULTY GRADIENT)

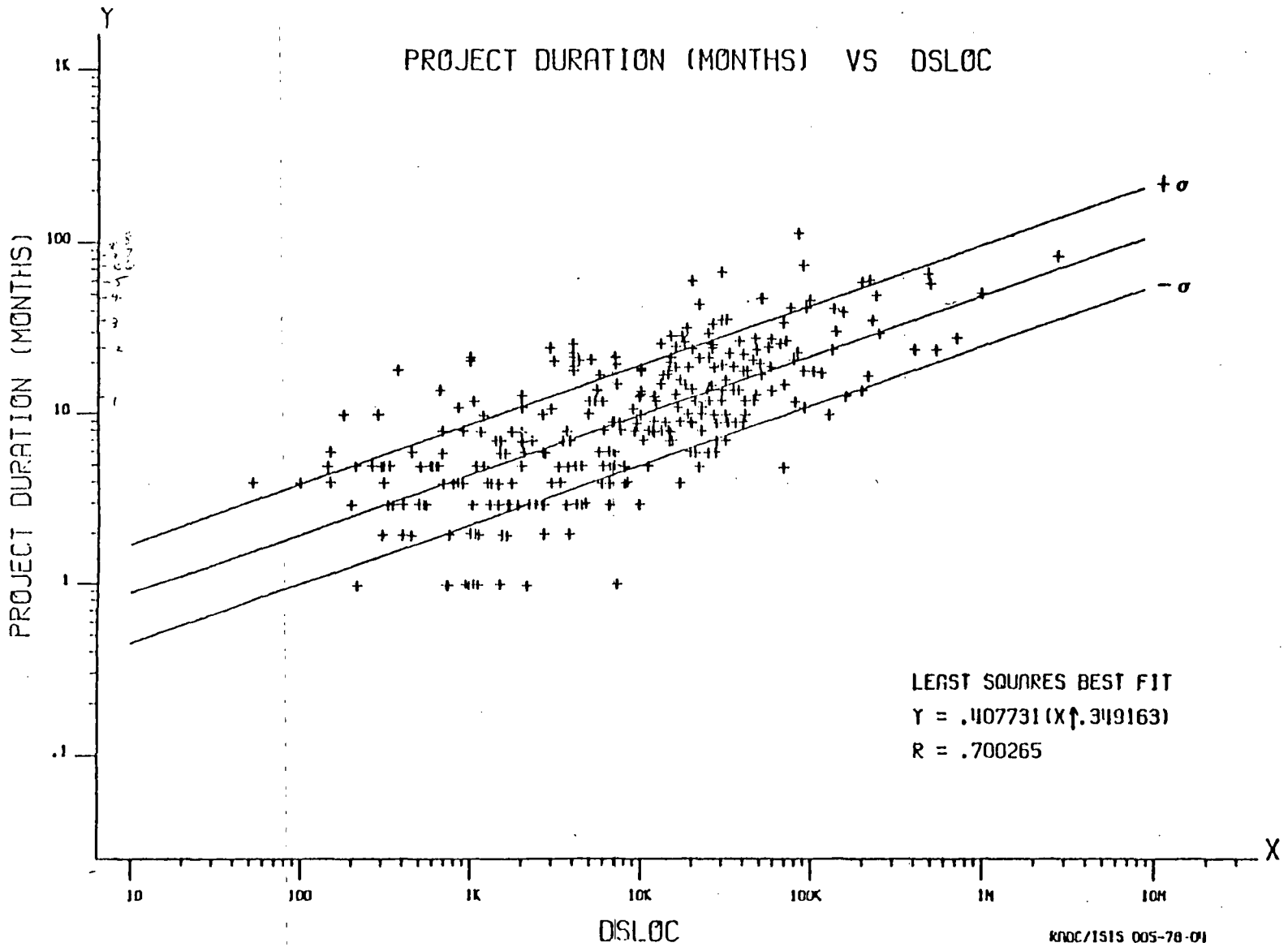
ACTUAL DATA

- DATA STREAM FROM PROJECT WHEN UNDERWAY TO DYNAMICALLY CONVERGE TO TRUE SYSTEM BEHAVIOR

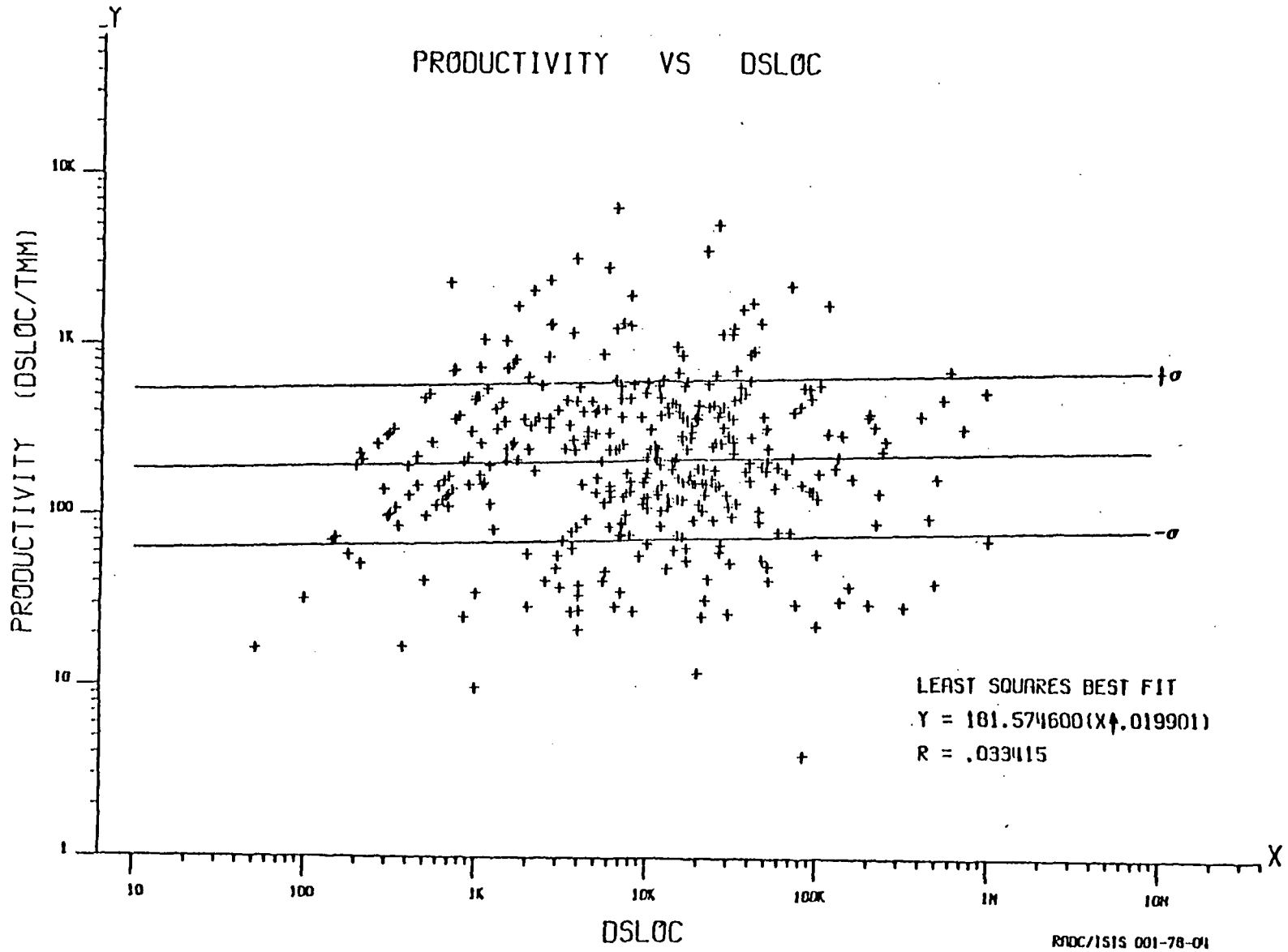
THE LIFE CYCLE METHOD
CAN ANSWER THE MANAGEMENT
QUESTIONS:

- CAN I DO IT?
- HOW MANY DOLLARS?
- HOW LONG?
- HOW MANY PEOPLE?
- WHAT'S THE TRADE OFF?
- WHAT'S THE RISK?

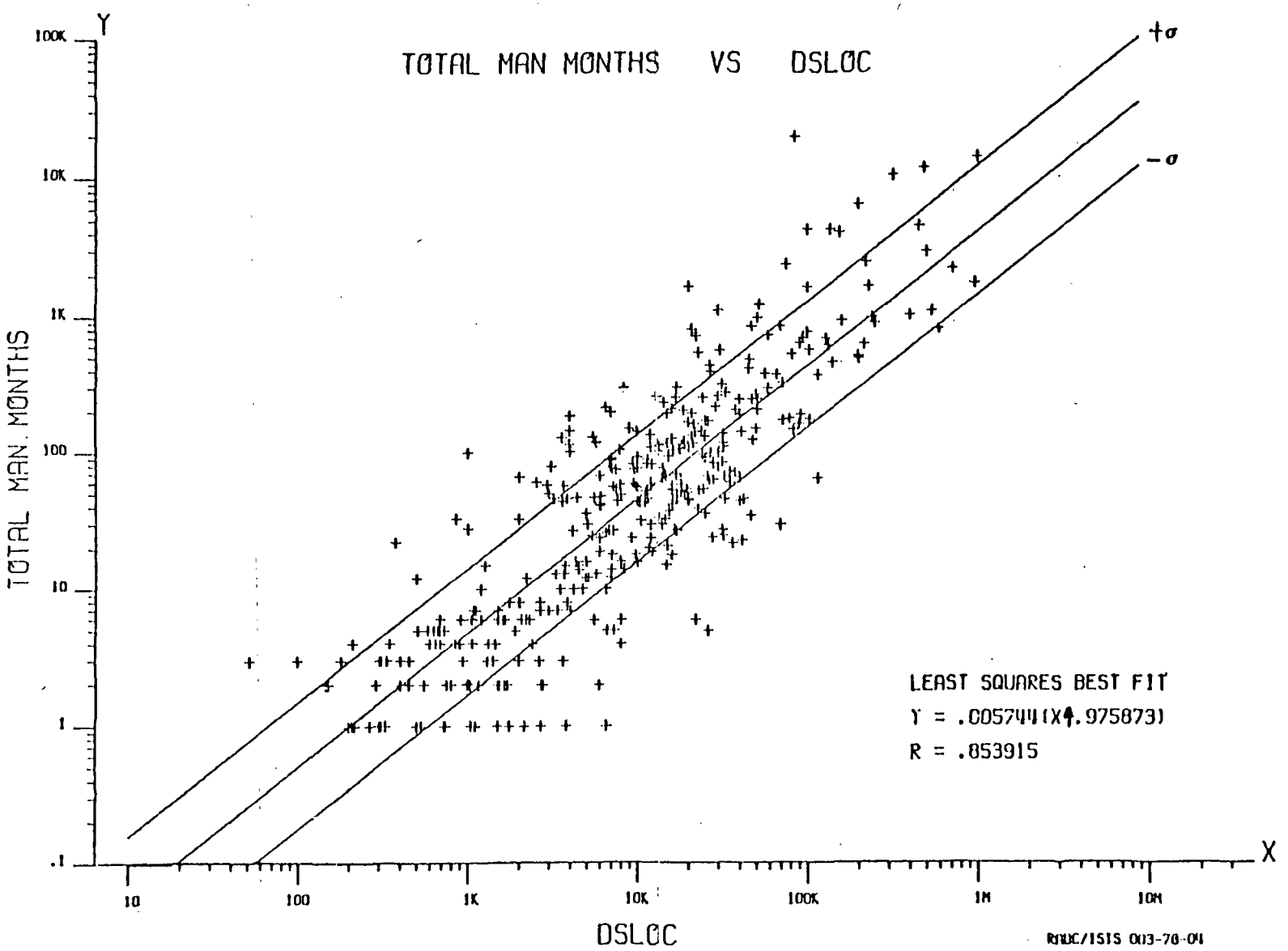
L. Putnam
QSM, INC.

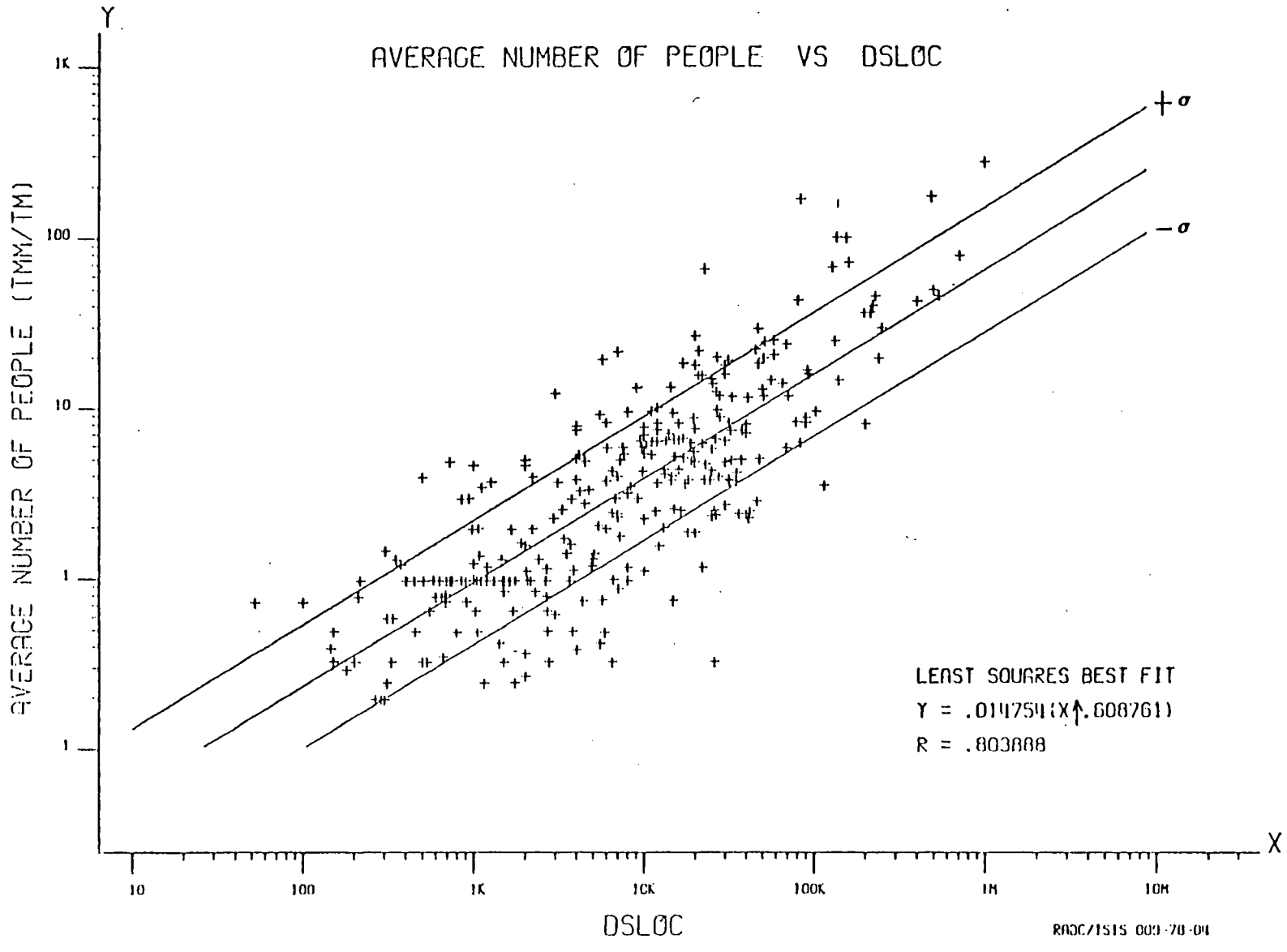


L. Putnam
QSM, INC.
Page 37 of 40



L. Putnam
QSM, INC.


TOTAL MAN MONTHS VS DSLOC
Y
X
TOTAL MAN. MONTHS
DSLOC
100K
10K
1K
100
10
1
.1
10
100
1K
10K
100K
1M
10M
+σ
−σ
LEAST SQUARES BEST FIT
Y = .0057411(X↑.975873)
R = .853915
RADC/ISIS 003-70-04



L. Putnam
QSM, INC.

Page intentionally left blank

Page intentionally left blank

omit
To
P. 74

PREDICTING PROGRAMMER'S PERFORMANCE FROM MODELS OF SOFTWARE COMPLEXITY

Sylvia B. Sheppard
Information Systems Programs
General Electric Company
Arlington, Virginia

The research reported here was designed to investigate factors influencing two tasks in software maintenance: understanding an existing program and implementing modifications to it. These factors included structured programming techniques, cognitive programming aids, and program complexity. While the first two factors were manipulated experimentally, no systematic attempt was made to manipulate program complexity.

Three software complexity metrics (number of statements, McCabe's v(G), and Halstead's E) were compared to performance on two software maintenance tasks. In an experiment on understanding, length and v(G) correlated with the percent of statements correctly recalled. In an experiment on modification most of the significant correlations were obtained with metrics computed on modified rather than unmodified code. All three metrics correlated with time to complete the modification, while only length and v(G) correlated with the accuracy of the modification. Relationships in both experiments occurred in unstructured rather than structured code, and in the second experiment primarily where no comments appeared in the code. The metrics were also most predictive of performance for inexperienced programmers. Thus, these metrics appeared to assess psychological complexity only where programming practices did not provide assistance in understanding the code.

Assessment of the psychological complexity of software appears to require more than a simple count of operators and operands or basic control paths. Many programs have characteristics unassessed by these metrics which may heavily influence psychological complexity. For instance, the use of structured coding techniques or comments reduces the cognitive load on a programmer in ways unassessed by the complexity metrics. Further, complexity metrics may not be capturing the most important constructs for predicting the performance of experienced programmers who may either be conceptualizing programs at a level other than that of operators, operands, and basic control paths, or who can fit the program into a schema similar to one with which they have had previous experience.

Further work in the area of psychological complexity should identify a set of cognitive psychological principles relevant to programming tasks. Metrics could then be developed which assess the qualities of software which are most closely related to these principles. Such an exercise might not only lead to improved metrics for assessing psychological complexity but might also identify some programming practices which could lead to simplified, more easily maintained software.

ACKNOWLEDGEMENT

The research reported in this paper was supported by Contract No. N00014-77-C-0158 Engineering Psychology Programs, Office of Naval Research. However the opinions expressed in this paper are not necessarily those of ONR or the Department of Defense.

Page intentionally left blank

Page intentionally left blank

SOFTWARE RELIABILITY MODELLING IN FEDERAL SYSTEMS DIVISION

W. Douglas Brooks
Software Engineering
and Technology

BACKGROUND

- EMPHASIS ON SOFTWARE QUALITY
  - DoD
  - Other Customers
  - Internal
- NEED FOR QUANTITATIVE APPROACH
  - Definition of Reliability
  - Data Collection and Analysis

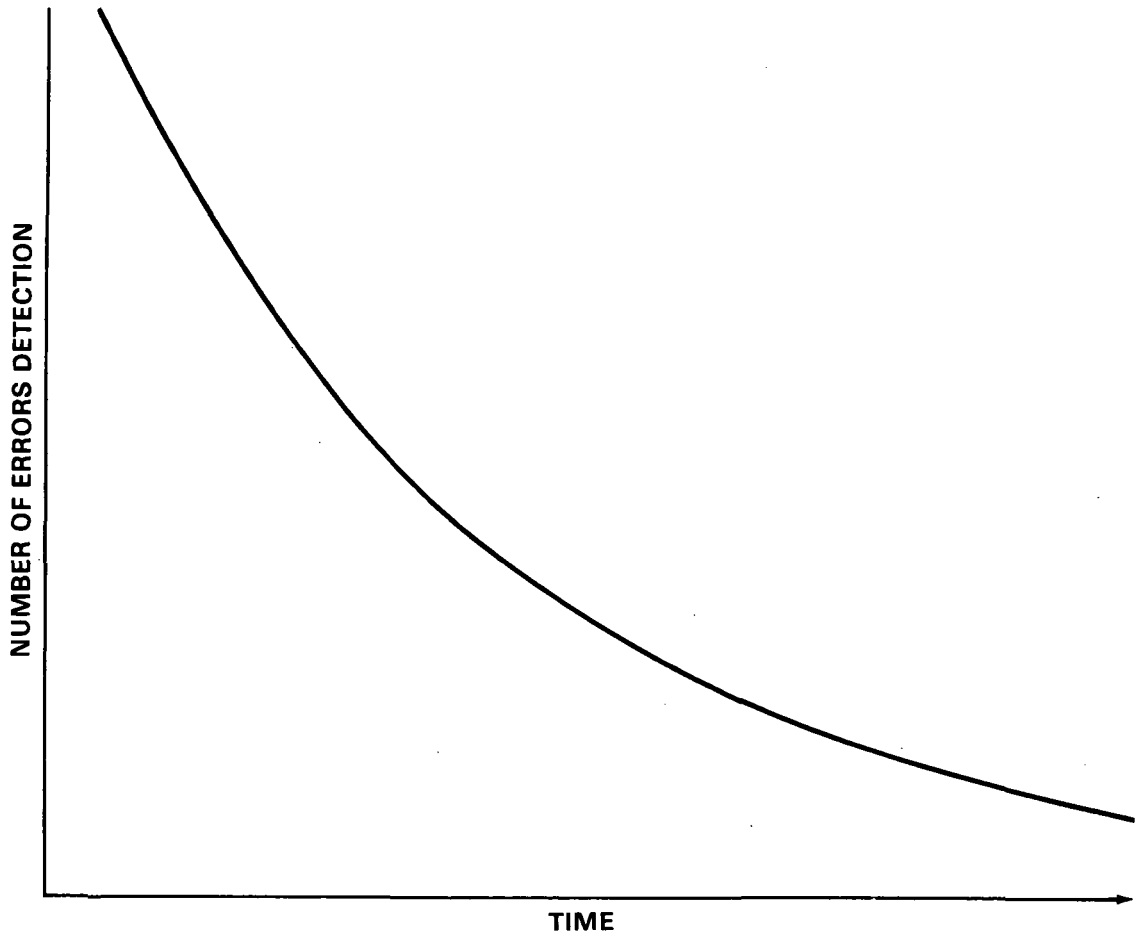
W. Brooks
IBM

PREDICTION FROM ERROR HISTORY

- ERRORS DETECTED ARE PROPORTIONAL TO REMAINING ERRORS
- PROPORTIONALITY FACTOR IS CONSTANT
- ERRORS ARE CORRECTED WITHOUT FURTHER ERRORS
- EQUAL TEST TIMES FROM ONE OCCASION TO THE NEXT

W. Brooks
IBM

THE IDEAL WORLD



$f(t) = \lambda e^{-\lambda t}$

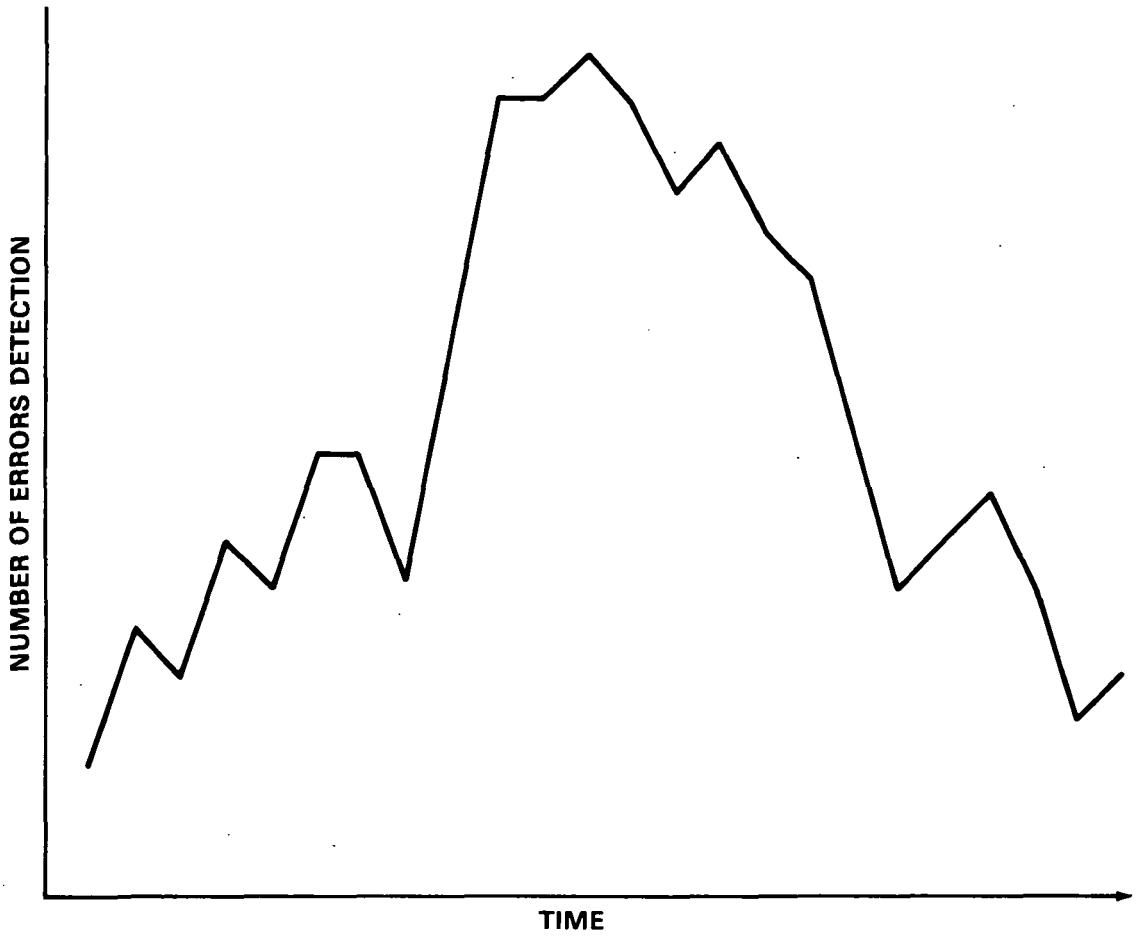
$R(t) = 2 - f^{t} e^{-\lambda t} dt = e^{-\lambda t}$

λ = NUMBER ERRORS REMAINING TIMES A CONSTANT (ϕ)

MODEL ESTIMATES N_r AND ϕ

W. Brooks
IBM

THE REAL WORLD OF SOFTWARE DEVELOPMENT



$f(t_l, w_j, n)$

DEFINITION

RELIABILITY IS THE PROBABILITY THAT NO MORE THAN X ERRORS WILL OCCUR DURING SOME SPECIFIED FUTURE TIME INTERVAL, UNDER SPECIFIED TESTING CONDITIONS

W. Brooks
IBM

EXPLANATIONS OF IRREGULARITIES

RANDOM FLUCTUATIONS

NON-RANDOM FLUCTUATIONS

- SYSTEM IS BEING DEVELOPED AND TESTED INCREMENTALLY
- TESTING SCENARIOS – SOME MODULES NOT ALWAYS TESTED
- MODULES – EVEN THE SYSTEM – NOT TESTED EQUAL AMOUNTS FROM ONE OCCASION TO THE NEXT
- PROGRAMMERS DO MAKE ERRORS IN DEBUGGING
- CORRECTED ERRORS EXPOSE ADDITIONAL ERRORS TO DETECTION

IMPLICATIONS FOR MODELLING

CLASSICAL MODEL

$n_1^1 = N\phi$

$n_2^1 = (N - n_1)\phi$

$n_3^1 = (N - n_1 - n_2)\phi$

etc.

ADD PORTION OF THE SYSTEM UNDER TEST

$n_1^1 = W_1 N \phi$

$n_2^1 = (W_2 N - n_{12})\phi$

$n_3^1 = (W_2 N - n_{13} - n_{23})\phi$

etc.

ADD UNEQUAL TIME INTERVALS

$n_1^1 = W_1 N [1 - (1 - \phi)^{t_1}]$

$n_2^1 = (W_2 N - n_{12}) [1 - (1 - \phi)^{t_2}]$

$n_3^1 = (W_2 N - n_{13} - n_{23}) [1 - (_1 - \phi)^{t_3}]$

etc.

ADD REINSERTION/UNCOVERY RATE

$n_1^1 = W_1 N [1 - (1 - \phi)^{t_1}]$

$n_2^1 = [W_2 N - (1 - r) n_{12}] [1 - (1 - \phi)^{t_2}]$

$n_3^1 = [W_3 N - (1 - r)(n_{13} + n_{23})] [1 - (1 - \phi)^{t_3}]$

etc.

W. Brooks
IBM

IMPLICATIONS FOR DATA COLLECTION

FOR EACH TEST OCCASION

- Date
- Number of software errors (not incidents)
- Number of errors by module/subsystem
- Type of software error
- Amount of test time by module/subsystem
- Number of source instructions by module/subsystem

OUTPUTS OF THE MODEL

- Estimated number of errors remaining
- Probability of no more than X errors in specified time
- Estimated probability of error detection
- Amount of additional time required to achieve specified reliability
- Measure of confidence in goodness of fit

W. Brooks
IBM

PROBLEMS IN SOFTWARE ERROR DEFINITION

OR

AN ERROR IS AN ERROR BUT
AN INCIDENT IS A

- Hardware Error
- Requirements Error
- Duplicate Report
- False Report
- Control Program Error
- Design Error
- Coding Error
  - One Symptom
  - Multiple Symptoms

W. Brooks
IBM

EXAMPLE OF APPLICATION OF MODEL

- C&C LAB – DWS TRAINER SOFTWARE DEVELOPMENT
- PERFORMANCE REQUIREMENTS: SYSTEM RELIABILITY TEST. 48 HOURS OF CONTINUOUS OPERATION. ERRORS ALLOWED:
  - CATEGORY I – MAXIMUM OF 1
  - CATEGORY II – MAXIMUM OF 3
  - CATEGORY III – MAXIMUM OF 13
- DATA COLLECTION REQUIREMENTS TO USE MODEL SPECIFIED
- INDEPENDENT ANALYSIS SHOWS THAT TO BE REASONABLY CERTAIN OF MEETING REQUIREMENTS, CAPABILITIES SHOULD BE:
  - CATEGORY I – .5 ERRORS/48 HOURS
  - CATEGORY II – 1.8 ERRORS/48 HOURS
  - CATEGORY III – 9.5 ERRORS/48 HOURS

MODEL APPLIED TO THIS PROBLEM

- DURING DEVELOPMENT RECORD APPROPRIATE ERRORS AND EFFORT
- MAKE PREDICTIONS THROUGHOUT DEVELOPMENT AND TESTING
- USE INCENTIVE AWARDS TO PERFORM TRADE-OFFS
- MAKE RECOMMENDATIONS TO MANAGEMENT
  - NEED FOR ACCELERATED TESTING
  - TEST STRATEGIES
- VALIDATE PREDICTIONS
- DERIVE IMPLICATIONS AND DATA REQUIREMENTS FOR USE OF MODEL
  - DESIGN REVIEW
  - CODE INSPECTION
  - UNIT TEST
  - INTEGRATION TEST

W. Brooks
IBM

PANEL #3

MEASURING SOFTWARE DEVELOPMENT METHODOLOGIES

Chairperson . . . . . Marv Zelkowitz (University of Md)

Member #1 . . . . . Bob Reiter (University of Md)

Member #2 . . . . . Phil Milliman (General Electric)

Member #3 . . . . . Paul Scheffer (Martin Marietta Corporation)

D3

INVESTIGATING SOFTWARE DEVELOPMENT APPROACHES: A SYNOPSIS *

Robert W. Reiter, Jr.
Department of Computer Science
University of Maryland
College Park, Maryland 20742

INTRODUCTION

The paper reports on research comparing various approaches, or methodologies, for software development. The study focuses on the quantitative analysis of the application of certain methodologies in an experimental environment, in order to further understand their effects and better demonstrate their advantages in a controlled environment. A series of statistical experiments were conducted

The paper reports on research comparing various approaches, or methodologies, for software development. The study focuses ontthe quantitative analysis of the application of certain methodologies in an experimental environment, in order to further understand their effects and better demonstrate their advantages in a controlled environment. A series of statistical experiments were conducted, comparing programming teams which used a disciplined methodology (consisting of top-down design, process design language usage, structured programming, chief programmer teams, and code reading) with programming teams and individual programmers which employed their own ad hoc approach. Specific details of the experimental setting, the investigative approach (used to plan, execute, and analyze the experiments), and some of the results of the experiments are discussed.

The purpose of the research was to develop an investigative methodology for experimentally studying and quantitatively characterizing the effect of methodologies and programming environments on software development. It involves the quantitative measurement and analysis of both the process and the product of software development, in manner which is minimally obstrusive (to those developing the software), very objective, and highly automatable. The basic premise is that distinctions among the groups exist both in the process and in the product.

SPECIFICS

Nineteen units (teams or individuals) each performed the same software development task, but under controlled and slightly varied conditions. Two programming factors, size of programming team and degree of methodological discipline, each with two levels (single individual, and three-person team; the ad hoc approach, and the disciplined methodology), were chosen as the independent variables and formed the experimental treatments. The dependent variables to be observed and measured were a large set (over 125) of programming aspects. The teams and individuals were

*Research supported in part by the Air Force Office of Scientific Research grant AFOSR-77-3181A to the University of Maryland. Computer time supported in part through the facilities of the Computer Science Center of the University of Maryland.

placed into three treatment groups, designated A, B and C (of 6, 6 and 7 units, respectively), each operating under a certain combination of factor-levels:

A – individuals, ad hoc approach;

B – three-person teams, ad hoc approach;

C – three-person teams, disciplined methodology.

The time and place for the experiment was Spring, 1976, in conjunction with two academic courses at the University of Maryland. The particular project or application to be developed was compiler for a small high-level language and a simple stack machine. This task was roughly a two man-month effort, and the resulting software systems averaged about 1200 source lines or 600 executable statements, in high-level structured-language code. The participants were advanced undergraduates and graduate students in the Computer Science Department. The implementation language was the high-level structured-programming language SIMPL-T [Basili and Turner 76], which is used extensivly in course work at the University and has string-processing capabilities similar to PL/1.

Data collection for the experiment was automated on-line, with essentially no interference to the programmer's normal pattern of actions during computer sessions. Special module compilation and program execution processors created an historical data base of source code and test data accumulated throughout the project development. Scores corresponding to each of the programming aspects were extracted directly and algorithmically from this data base.

The programming aspects represent specific automatically isolatable and observable features of the programming phenomenon, related to either the product or the process of software development. Product aspects are based on the syntactic content and organization of the symbolic source code which represents the complete final product developed. Process aspects are related to characteristics of the development process itself, in particular, the cost and required effort as reflected in the number of computer job steps (or runs) and the amount of textual revision of source code during development. Major headings for the particular programming aspects reported on in this study are listed in the accompanying table, with qualifying subcategories mentioned in square brackets.

APPROACH

The investigative methodology was designed and developed as a scientific and empirical solution to the problem of comparing software development efforts under various conditions. It was used to guide the planning, execution, and analysis of the set of experiments which comprise this study. The approach consists of eleven steps or elements, as shown in the accompanying schmatic diagram which charts the general flow (solid lines) and some of the interrelationships (dashed lines) among these elements.

The methodology begins with Questions of Interest, which are turned into Research Hypotheses and Statistical Hypotheses. The Statistical Model is very important since it governs the Experimental Design and several other elements. Statistical Results, corresponding directly to the Statistical Hypotheses, are determined by the Colledted Data via the Statistical Test Procedures. Research Framework(s) are necessary to organize the large volume of hypotheses and results into a smaller, more managable form as Statistical Conclusions and Research Interpretations.

RESULTS

The methodology provides that the study's results be separated into statistical conclusions, representing factual findings, and research interpretations, representing intuitive judgements.

For each aspect there is one statistical conclusion which states any differences observed among the three programming environments represented by the groups A, B, and C. These outcomes are expressed in the form of "equations"; e.g., A<B=C means that the average score for the individual programmers was appreciably lower than the average scores for the ad hoc teams and the disciplined teams which both had about the same average score. In addition to the null outcome (A=B=C) of no observed differences, there were twelve other possible outcomes, as noted in the accompanying table. The table simply lists all the non-null conclusions, arranged by outcome. The values in the "error" column state the risk, as a probability value, of erroneously making that conclusion and indicate how strongly pronounced the differences were in the data. Although there is much fascinating material in these findings, space permits only a few particularly interesting conclusions to be pointed out.

The A<B=C outcome was quite pronounced for the SEGMENTS aspect, indicating that the individuals built their systems with fewer routines on the average than either the ad hoc teams or the disciplined teams, which used about the same number of routines. According to the A<B=C and B=C<A outcomes, the individuals had noticably less global variables and more local variables than both types of teams. The C=A<B outcomes for IF statements and DECISIONS indicate aspects where the disciplined teams behaved like the individuals and both were different than the ad hoc teams. For the number of COMPUTER RUNS (JOB STEPS), and several subcategories, the C < A =B outcomes have very low error risks and indicate that the disciplined teams out-performed both the individuals and the ad hoc teams in these aspects. On the number of PROGRAM CHANGES – a measure of the amount of cummulative textual revision of the program source code during development, which has been shown to correlate well with total error occurrences [Dunsmore and Gannon 77] – the same data scores which support the C<A<B conclusion at a high risk of error (0.185) also support the C < A = B conclusion at a very low risk of error (0.004), indicating a strong distinction in terms of error-prone-ness in favor of the disciplined teams.

One framework for the interpretation of these conclusions is the concept of how the disciplined methodology actually impacts the software development process and product. Prior to conducting the experiment, certain general beliefs (see details on accompanying slide) about the impact had

Robert W. Reiter, Jr.
University of Maryland

been formulated. Certain basic suppositions (a priori expectations), for how the experiments should turn out if the beliefs were true, were constructed from the general beliefs. Examination of how the conclusions stack up against the suppositions (how true the beliefs are) shows that none of the conclusions for any of the observed programming aspects contravene the basic suppositions. Thus, the study's results may be interpreted as strong experimental evidence in favor of these general beliefs.

SUMMARY

A practical methodology was designed and developed for experimentally and quantitatively investigating the software development phenomenon. It was employed to compare three particular software development environments and to evaluate the relative impact of a particular disciplined methodology (made up of so-called modern programming practices). The experiments were successful in measuring differences among programming environments and the results support the claim that disciplined methodology effectively improves both the process and product of software development. The results will be used to guide further experiments and will act as a basis for analysis of software development products and processes in the Software Engineering Laboratory at NASA/GSFC [Easili et al. 77]. The intention is to persue this type of research, especially extending the study to include more sophisticated and promising programming aspects, such as Halstead's software science quantities [Halstead 77] and other software complexity metrics [McCabe 76].

REFERENCES

1. [Basili and Reiter 78] V.R. Basili and R.W. Reiter, Jr. Investigating Software Development Approaches. Technical Report TR-688, Department of Computer Science, University of Maryland, August, 1978.

2. [Basili and Turner 76] V.R. Basili and A.J. Turner. SIMPL-T, A Structured Programming Language. Paladin House Publishers, Geneva, Illinois. 1976.

3. [Basili et al. 77] V.R. Basili, M.V. Zelkowitz, F.E. McGarry, R.W. Reiter, Jr., W.F. Truszkowski, and D.L. Weiss. The Software Engineering Laboratory. Technical Report TR-535, Department of Computer Science, University of Maryland. May, 1977.

4. [Dunsmore and Gannon 77] P.E. Dunsmore and J.D. Gannon. Experimental Investigation of Programming Complexity. Proceedings of ACM-NES Sixteenth Annual Technical Symposium: Systems and Software (June 1977), Washington, D.C., pp. 117-125.

5. [Halstead 77] M. Halstead. Elements of Software Science. Elsevier Computer Science Library. 1977.

6. [McCabe 76] T.J. McCabe. A Complexity Measure. IEEE Transactions on Software Engineering, Vol. 2, No. 4 (December 1976), pp. 308-320.

Programming Aspects

****

Development Process Aspects :

COMPUTER RUNS (JOB STEPS)
[compilations, executions, miscellaneous]
ESSENTIAL RUNS (JOB STEPS)
AVERAGE UNIQUE COMPILATIONS PER MODULE
MAX UNIQUE COMPILATIONS FOR ANY ONE MODULE
PROGRAM CHANGES

****

Final Product Aspects :

MODULES
SEGMENTS
SEGMENT TYPE COUNTS
[function, procedure]
SEGMENT TYPE PERCENTAGES
[function, procedure]
AVERAGE SEGMENTS PER MODULE

LINES

STATEMENTS
STATEMENT TYPE COUNTS
[:=, IF, CASE,WHILE, EXIT, (proc)CALL, RETURN]
STATEMENT TYPE PERCENTAGES
[:=, IF, .CASE, WHILE, EXIT, (proc)CALL, RETURN]
AVERAGE STATEMENTS PER SEGMENT
AVERAGE STATEMENT NESTING LEVEL

DECISIONS

FUNCTION CALLS
[non-intrinsic, intrinsic]

TOKENS
AVERAGE TOKENS PER STATEMENT

INVOCATIONS
[function, procedure; non-intrinsic, intrinsic]
AVG INVOCATIONS PER (CALLING) SEGMENT
[function, procedure; non-intrinsic, intrinsic]
AVG INVOCATIONS PER (CALLED) SEGMENT
[function, procedure]

DATA VARIABLES
DATA VARIABLE SCOPE COUNTS
[global, parameter,local]
DATA VARIABLE SCOPE PERCENTAGES
[global, parameter, local]
AVERAGE GLOBAL VARIABLES PER MODULE
[modified, not modified; non-entry, entry]
AVERAGE NON-GLOBAL VARIABLES PER SEGMENT
[parameter, local]

PARAMETER PASSAGE TYPE PERCENTAGES
[value, reference]

(SEG,GLOBAL) ACTUAL USAGE PAIRS
[modified, not modified; non-entry, entry]
(SEG,GLOBAL) POSSIBLE USAGE PAIRS
[modified, not modified; non-entry, entry]
(SEG,GLOBAL) USAGE RELATIVE PERCENTAGES
[modified, not modified; non-entry, entry]

(SEG,GLOBAL,SEG) DATA BINDINGS
[actual; possible; relative percentage]

****

Robert W. Reiter, Jr.
University of Maryland

Robert W. Reiter, Jr.
University of Maryland

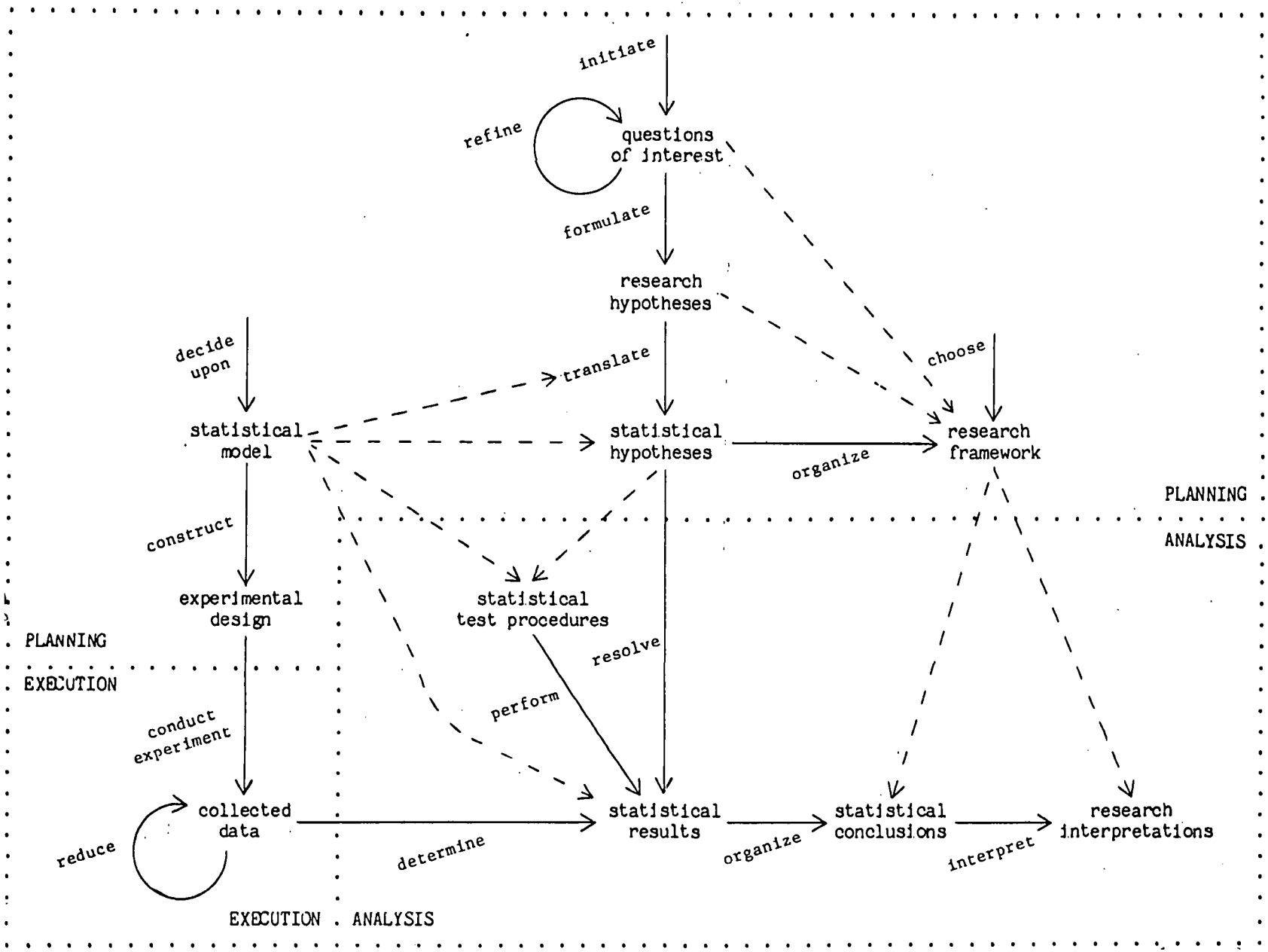


Diagram 1. Approach Schematic

Non-Null Conclusions, arranged by outcome

outcome	error freq	programming aspect
A < B = C	9	
	0.0634	SEGMENTS
	0.0698	DATA VARIABLES
	0.1476	DATA VARIABLE SCOPE COUNTS \ GLOBAL
	0.1614	DATA VARIABLE SCOPE COUNTS \ GLOBAL \ MODIFIED
	0.2015	DATA VARIABLE SCOPE COUNTS \ NON-GLOBAL
	0.1271	DATA VARIABLE SCOPE COUNTS \ NON-GLOBAL \ PARAMETER
	0.1507	DATA VARIABLE SCOPE PERCENTAGES \ NON-GLOBAL \ PARAMETER
	0.1748	AVERAGE NON-GLOBAL VARIABLES PER SEGMENT \ PARAMETER
	0.1227	(SEG,GLOBAL) POSSIBLE USAGE PAIRS
B = C < A	5	
	0.1706	AVERAGE STATEMENTS PER SEGMENT
	0.1699	AVG INVOCATIONS PER (CALLING) SEGMENT \ NON-INTRINSIC
	0.1699	AVG INVOCATIONS PER (CALLED) SEGMENT
	0.1936	AVG INVOCATIONS PER (CALLED) SEGMENT \ FUNCTION
	0.1090	DATA VARIABLE SCOPE PERCENTAGES \ NON-GLOBAL \ LOCAL
B < C = A	3	
	0.2195	STATEMENT TYPE PERCENTAGES \ CASE
	0.2364	(SEG,GLOBAL) USAGE RELATIVE PERCENTAGES
	0.1546	(SEG,GLOBAL) USAGE RELATIVE PERCENTAGES \ NOT MODIFIED \ NON-ENTRY
C = A < B	11	
	0.2134	SEGMENT TYPE COUNTS \ FUNCTION
	0.2321	STATEMENTS
	0.0780	STATEMENT TYPE COUNTS \ IF
	0.1732	STATEMENT TYPE COUNTS \ (PROC)CALL \ INTRINSIC
	0.0196	STATEMENT TYPE COUNTS \ RETURN
	0.1038	STATEMENT TYPE PERCENTAGES \ IF
	0.2065	STATEMENT TYPE PERCENTAGES \ RETURN
	0.1468	DECISIONS
	0.1732	INVOCATIONS \ PROCEDURE \ INTRINSIC
	0.0435	INVOCATIONS \ INTRINSIC
	0.1861	(SEG,GLOBAL,SEG) DATA BINDINGS \ POSSIBLE
C < A = B	8	
	0.0036	COMPUTER RUNS (JOB STEPS)
	0.0223	COMPUTER RUNS (JOB STEPS) \ MODULE COMPILATIONS
	0.0110	COMPUTER RUNS (JOB STEPS) \ MODULE COMPILATIONS \ UNIQUE
	0.0221	COMPUTER RUNS (JOB STEPS) \ PROGRAM EXECUTIONS
	0.1445	COMPUTER RUNS (JOB STEPS) \ MISCELLANEOUS
	0.0037	ESSENTIAL RUNS (JOB STEPS)
	0.0883	AVERAGE UNIQUE COMPILATIONS PER MODULE
	0.1180	MAX UNIQUE COMPILATIONS FOR ANY ONE MODULE
A = B < C	0	
A < B < C	0	
A < C < B	1	
	0.1194	LINES
B < C < A	2	
	0.1232	(SEG,GLOBAL) USAGE RELATIVE PERCENTAGES \ MODIFIED \ ENTRY
	0.1173	(SEG,GLOBAL) USAGE RELATIVE PERCENTAGES \ ENTRY
B < A < C	0	
C < A < B	1	
	0.1848	PROGRAM CHANGES
C < B < A	0	

Robert W. Reiter, Jr.
University of Maryland

Research Interpretations

General Beliefs:

- The disciplined methodology reduces the average cost and complexity of the process.
- The disciplined methodology can enable a programming team to compensate for their inherent coordination overhead and behave more like an individual programmer in terms of designing and building the product.

Basic Suppositions:

- on process aspects: $C \leqslant A, B$
- on product aspects: $A \leqslant C \leqslant B$ or $B \leqslant C \leqslant A$

Support from the conclusions:

- process:
  - $C < A = B$ on 8 aspects
  - $C < A < B$ on 1 aspect
  - $A = B = C$ on 1 aspect
- product:
  - $A < B = C$ on 9 aspects
  - $A = C < A$ on 5 aspects
  - $B < C = A$ on 3 aspects
  - $C = A < B$ on 11 aspects
  - $A < C < B$ on 1 aspect
  - $B < C < A$ on 2 aspects
  - $A = B = C$ on 96 aspects

None of the conclusions for any of the observed programming aspects contravene these basic suppositions.

Thus, the study's results may be interpreted as strong experimental evidence in favor of these general beliefs.

Page intentionally left blank

Page intentionally left blank

OMIT

MEASUREMENT AND EVALUATION OF MODERN PROGRAMMING PRACTICES: A CASE STUDY

Phil Milliman
Information Systems Programs
General Electric Company
Arlington, Virginia

In research sponsored by the Rome Air Development Center, a four man year project was studied to determine the effect of modern programming practices on the life cycle of a project. Analysis of errors and error rates, the source code, and development characteristics indicated that while the project was not greatly different from the standard industry project, it was consistently different and higher quality than a similar project in the same development environment.

Halstead's theory of software science was used in a number of different approaches to point out characteristics of the code and the performance of the project. In particular, the number of software problem reports was predicted, time to program the system, the program level, and the language level were examined. The project using modern programming practices had a higher program and language level than the similar project in the same environment. Other factors, such as McCall's quality metrics and comparison to the RADC data base supported these findings.

Project development data were examined, and error rates were used to predict the number of software problem reports. Error rates and the number of lines changed per run decreased over the coding period.

It was concluded that modern programming practices do make a positive difference in performance, but that other environmental considerations such as computer access, management practices, and type of problem may obscure their benefits when compared to projects in other development environments. The present study demonstrated that evaluation is possible in a matched projects environment. Matched projects allow much of the environment to be considered equal, with increased attention on the few differences being manipulated. In this situation a more objective analysis of a given programming practice may be obtained rather than a blanket judgement as in the present study.

REFERENCES:

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.

Gordon, R.D., & Halstead, M.H. An experiment comparing Fortran programming time with the software physics hypothesis. AFIPS Conference Proceedings, 1976, 45, 935-937.

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

McCabe, T.J. A complexity measure. IEEE Transactions on Software Engineering, Vol. SE-2, NO4, December 1976.

McCall, J.A., Richards, P.K., and Walters, G.F. Factors in software quality (Tech. Rep. 77C1502). Sunnyvale, CA: General Electric Corporation, Information Systems Programs, Command and Information Systems. Prepared for Electronic Systems Division, Air Force Systems Command and Rome Air Development Center, June 1977.

ACKNOWLEDGEMENT:

Research described in this abstract was supported by Contract No. F30602-77-C-0194 with Rome Air Development Center. However, opinions expressed in this paper are not necessarily those of RADC or the Department of Defense.

D4

ANALYTIC TECHNIQUES FOR EVALUATING SOFTWARE DESIGNS AND METHODOLOGIES

SEPTEMBER 1978

by:
Paul A. Scheffer

MARTIN MARIETTA AEROSPACE
DENVER, COLORADO

I. INTRODUCTION

In recent years several methodologies have been developed to assist in the software development process at the requirements-to-design engineering phase. The objective of many of these approaches is to assist the designer in deriving, from a given set of requirements, a modular framework for the system which can be associated with qualities such as adaptability to changes in requirements, testability, maintainability, interface correctness, etc. Further, in many cases these approaches are complemented by a language designed to support the basic concept of the technique. Systems such as PSL/PSA, SREM, SSL and strategies attributed to Dijkstra, Mills, Parnas, Jackson, and Myers fall into this class and are currently receiving much attention by the software engineering community. This attention can be attributed in part to the growing recognition that rigor at the requirements and design phases tends to minimize the costly "error days" associated with software. It also allows for manageability of evolving requirements and addresses the true life cycle costs of systems. This attention, however, has yet to lead to guidelines for an intelligent selection from this wave of methodologies. With some minor exceptions, the lack of quantitative and careful qualitative evaluation of benefits and costs associated with "front end" development strategies is notable.

In this paper, several analytic techniques are discussed as they might be applied to software design. Such techniques show promise on two fronts: the ability to quantitatively measure various aspects (viz. qualities) of a given design statement; and, by using quality indexes for several designs produced under different auspices, to yield a comparative assessment of individual strategies, techniques, or personnel. The rationale behind the comparision idea is the fact that the common denominator of all design strategies, at any level, is their treatment of the structural characteristics of problems, i.e., the systematic decomposition of the original problem into a logically organized set of subproblems which contribute to the ultimate objective. The assumption is made that attributes of design quality are sufficiently manifested in the structural characteristics of the design so that they can be measured by a static analysis. (Similar to measures of complexity for code.) Further, we believe an appropriate scheme for static analysis of structure may be completely adequate for comparing different strategies, and can also provide a general tool for the development of superior quality software.

The application of analytics in this paper is not dependent on either a specific methodology or the level of development to which it applies. Since any methodology has a "product" in which its influence is contained, we simply consider associating a measure with that product and hence provide for evaluation of different methodologies or design strategies.

II. METHODOLOGY REVIEW

A. Background

Many attempts have been made at defining the crucial aspects of software development, both in terms of the finished product, viz. the design, and the activity itself, the design process. By looking at finished designs, one hopes to be able to generalize the characteristics that differentiate the better products. The problem then is determining how to effect the desirable result – the good design – within the design process. Lacking a methodology which assures a quality design. about all one is left with is an iteration and assessment cycle on designing and its end result until a satisfactory development is achieved.

A methodology for design is generally applicable to a particular phase of the total design process. As we learn more and more about software phenomena, methodology principles are discovered which can be applied earlier and earlier in the design activity. (Compare for example the precepts of structured programming from a few years ago, to Dijkstra's levels of abstraction, hierarchical systems and families, and the "top-down" approach.) Hence we now have methodologies for requirements analysis, functional analysis, and system structuring as well as detailed design and programming. However, as we shall discuss in the next section, the study of these is dependent on the manner in which they are expressed. In the short methodology review which follows, the use of design languages should be envisioned in which the design concepts can be expressed.

B. Jackson – Data Structure Orientation

The Jackson approach (7) to software design considers the input data structure as the driving force to program design. The program is viewed as the means by which input data are transformed to output data. By paralleling the structure of the input and output data, the analyst can presumably be assured of a quality design, at least if "quality" data structures exist a priori. This approach implicitly relies on the rationality of the data structures used and acknowledges a restriction to sequential files and applications which do not require a DBMS.

C. Myers and Constantine – Composite Design

Although this approach (10, 20, 23) deals more with programming principles than overall software development, the ideas can readily be generalized. In considering the design of modules, Myers differentiates the attribute of structure from those of function (purpose) and performance (behavior):

> "Structure is a description of the construction of a program, in such terms as coding structure, module structure, task (parallel-process) structure, memory layout, and module interfaces."

The composite design approach combines the structure aspects of modules, data, tasks, and their interfaces.

The methodology influences the development of programs in that it directs an iterative decomposition of structural components, called composite analysis, using defined principles of strength and coupling. Designing affects the organization of modules and module elements which produces the amount of "related-ness" among elements. The degree to which elements in the same module are related is called strength, and the methodology calls for maximizing this factor. Relationships between elements in different modules measure coupling, and are to be minimized. For any given program design, the degree to which these principles are adhered to can be measured as program stability in terms of a probability value on the impact of changes.

D. Parnas – Information Hiding

On a more abstract level, David Parnas (12) has presented a way of designing software to produce a system structure which is more stable in the sense of being adaptable and flexible when it comes to localizing the effects of a change. The Parnas methodology is based on the concept of "information hiding". The design decomposition criteria concentrates on the decisions which must be addressed in the software development process. The decomposition method entails the maximal containment of information by each module so that its interface and definition reveals as little as possible about its inner workings. Modules, i.e., software systems, developed in this manner will have a decomposition structure much less mechanically arrived at.

With this approach, modules that are components of the same system know only what is necessary about one another, no more and no less. An important aspect here is the emphasis on the hierarchical organization of a system and the levels of abstraction which alone helps to minimize module dependencies and complexities. By ordering levels from the most primitive to the most abstract, each level appears as a virtual machine for the level above it. This in itself restricts the information a module on one level needs about the operation of modules on lower levels.

III THE ANALYSIS STRATEGY

A. Approach

The study of design strategies is dependent on two important factors. One is that some "tangible" or machine readable representation of a problem must be available which lends itself to use computer aids in studying strategies. The second is that a "canonical form" of structural representation can be derived upon which certain assessment metrics can be based. These two factors are generally realized by high level design languages and (tree) graphs. Most analytic techniques rely on the ability to transform any methodology "product" to an equivalent graph structure of nodes and links representing design elements and element relationships. This is especially true when considering aspects of structure and structural decomposition. However, there are less complex forms of analysis which can be derived directly from the syntactic constructs provided by a formal expression medium.

In Figure 1, a model for studying design strategies is presented. It basically summarizes the concepts and relationships involved and can be used as a focal point for the discussion which follows. The process described in Figure 1 reflects the assignment of a measurement characteristic as a goal for a selected problem statement or expression. The center of the figure identifies the analytic techniques which can be used – applied to a given expression to measure a specific characteristic. The bottom of the figure illustrates this for the older source code analysis technology. The majority of the techniques listed are based on the notion of structural decomposition. The decomposition problem considers the partitioning of a set, in which "interdependencies" among elements have been defined, into a collection of subsets (clusters) characterized by:

1. Strong interdependencies among elements with a cluster.

2. Weak interdependencies between elements in different clusters.

The decomposition problem is readily handled by using a tree graph model of the set which sonsiders interdependencies as node links (edges) and connected sub-trees for the partioning. The characterization then appears as a strength and coupling measure on the clusters.

B. Expression

The expression of a given problem can take many "forms", depending on the vehicle used to state it. In traditional software developments this has simply been natural language. However, it is the recent availability of specialized tools – the specification and design languages and methodologies – which allows formalized approaches to expression. This in turn is what has provided us with machine processible data representing some design level. It is the formalization provided by syntactically structured and unambiguous methods and vocabularies which provides this "data base".

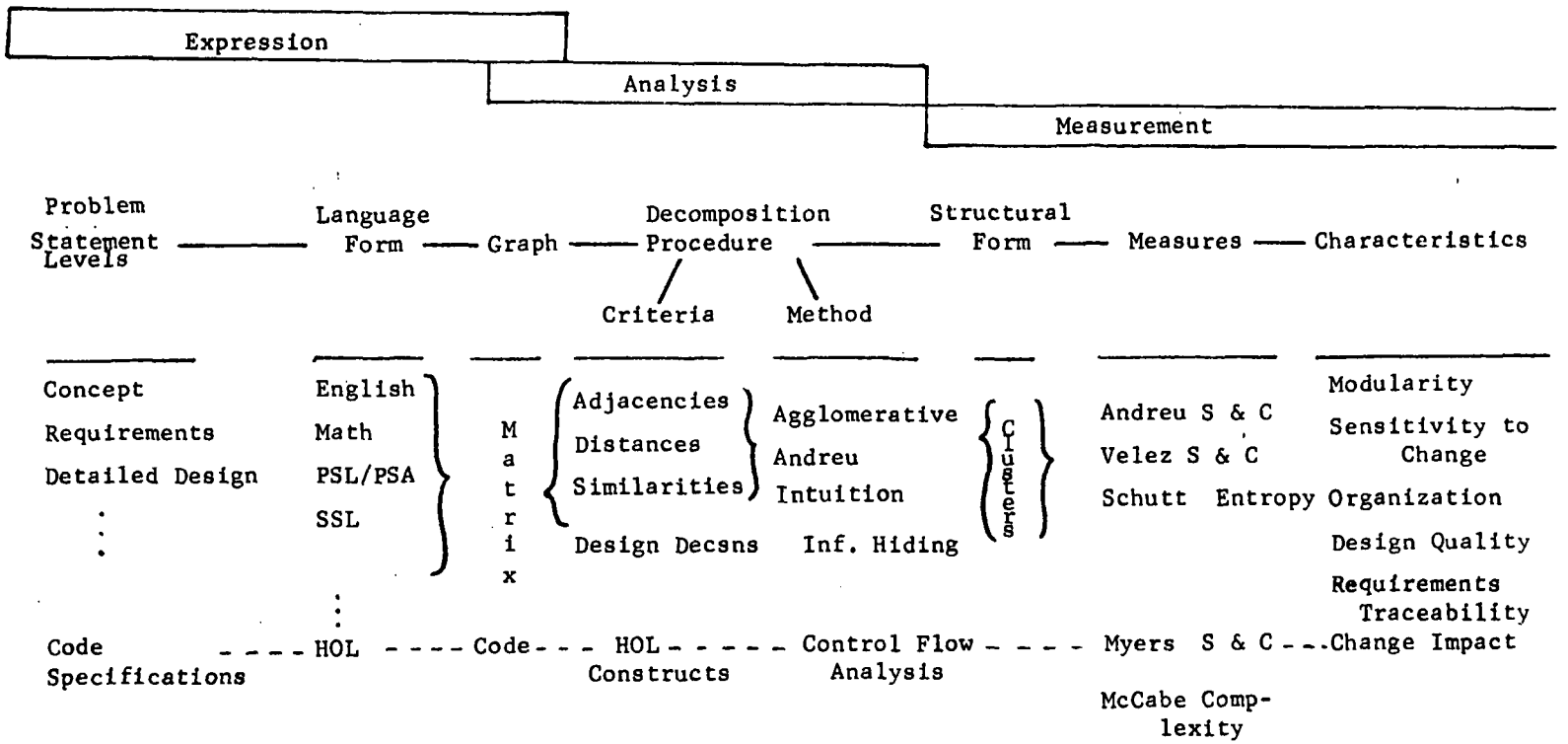


Figure 1. Formulation of the Study of Design Strategies

Several language forms are listed in Figure 1. Each has its peculiarities as to the design level it best expresses. For example, English is better for conceptualization, mathematics for algorithmic specifications, and Higher Order Languages for detailed design representation. The PSL/PSA scheme was originally intended as a documentation tool but continuing developments are making it widely applicable to many design levels. At Martin Marietta/Denver, we are developing a specialized requirements language (MEDL/R) to fill what we feel is a void for that level.

C. Analytic Techniques

1. The Andreu Approach – In (2), Andreu describes how the graph decomposition problem can be approached with both classical and hueristic cluster analysis techniques. Algorithmic schemes are based on a matrix of coefficients which be an array of adjacencies, minimum-path distances, or similarities; each provides a distance function representation of how a node is related to all the other nodes in a graph. Algorithmic methods are then described which use these metrics for the decomposition criterion in producing different structural clusterings of graph nodes.

The basic hueristic algorithm defines the "nearest-neighbor" set of nodes for each component of the graph. A distance count d_{ij} identifies the number of links between any two nodes. The distance metric used is what determines the meaning of "nearest". Hence for each node i, a neighborhood set is defined:

$$N_i = \left\{ n_j \mid d_{ij} \mid < T \right\}$$

Clusters are produced by considering the subsets for which $|N_i|$ is greater than some threshold parameter k. The heuristic assumption is that the code notes n_i of these k neighborhoods form "kernels" of importance over the entire graph. By forming successive intersections of neighborhoods and considering kernal clusters as nodes, the process becomes iterative. Inter-cluster linkages are determined, leaf nodes and "unimportant" clusters get merged (set union) and a decomposition is formed. A strength and coupling measure is used to evaluate a particular decomposition; strength is increased by node counts and linkages within a cluster, coupling is reduced by fewer inter-cluster linkages.

2. The Agglomerative Techniques – These are procedures which start with a set of n one-member clusters and try to reduce the number as dictated by some meaningful criteria. Typically, agglomerative techniques are measure independent in that they proceed until the number of clusters is reduced to one (the entire graph). The order in which elements are assigned to clusters that will eventually merge into the complete original set is then used to identify a "reasonable" set of clusters. Variations on these techniques are possible by using different decomposition criteria (e.g. dissimilarity matrix instead of similarity; various definitions of node-node distance). The major disadvantage of agglomerative techniques is that early decisions which categorize (cluster) a node cannot be changed at later stages – hence an initially poor assignment can never be reconsidered or recovered from. Also, some measure must be used to describe which of the interim partitionings is the most useful.

3. Static Analyzers – Another class of design decomposition schemes is based on the linguistic expression of a design. The ready availability of a "tangible" form of a design represented by its collection of "source" statements has prompted the development of many analysis tools analagous to source code analyzers. Most prominent in this activity are the measurement schemes which result in "complexity scores". These are developed from some static form of analysis of statements which generally assess:

- keyword occurrances and relationships

- module linkages, e.g., calling forms, parameter lists, common areas, etc.
- syntactic structures, e.g., assignments, DO's, CASE, etc.
- control structures, e.g., IF ....., CALL, Branches, etc.

Graph structures are not directly involved in these cases; a total decomposition (where each node is a cluster) is essentially assumed. Hence static analyzer programs use statistical and control flow analysis techniques to produce measures of quality, complexity, or structured-ness.

Most noteworthy in this area is the approach of Myers (10) who develops structure criteria based on probability measures of dependencies between modules. The probability measures are developed from applying module strength and coupling measures to design structures which are easily envisioned in the source code implementation. Even though heuristically formulated, this approach seems most practical in the determination of a software system's sensitivity to change brought about by simple maintenance of fluctuating requirements.

D. Measurement

The ability to compute a "quality of design" score is fundamental in the scheme for strategy comparison. Recent research has developed design quality metrics and measures from various viewpoints. Andreu (1) uses a strength and coupling measure applied to a graph representation of requirements and their inter-relationships for preliminary design. Myers (10) has developed a model in terms of probability measures applied to discrete strength and coupling factors of program modules which is used to assess the ramifications of making program changes. Schutt, et. al. (3,17) have applied an information entropy measure to hypergraph representations of computer processes and data structures. And, McCabe (8) shows a method for determining quality as a function of module "structured-ness". Each of these approaches relates in some way to a system measure.

We feel that the concept of subset strength and coupling (S&C) as used by Myers, Constantine, Andreu, et. al. is most appropriate as a quality measurement. The S&C concept assumes that it is the links which give structure to the entire graph. Consequently a structural evaluation of a partitioning involves the determination of how tightly coupled (linked) the nodes within a cluster area, as well as the extent to which two different clusters are related (number of linkages between).

In Figure 1 we listed several explicit ways of producing a "quality of design" score. While each of these has its own merits, the actual formulation of a quality measure is not as much at issue as what it means and what we can do with it. For example, the Andreu Strength and Coupling measure associates "goodness" with both modularity and sensitivity to change, in the sense that he derives his input from requirements statements and is concerned with the impact of fluctuating requirements. The Myers S&C measure is probability based using source code relationships as input. The interpretation here is that a programming change in any one module will affect all other parts with some probability as a function of how strongly the two modules are joined (coupled) and how isolated the changed module is (strength). The McCabe measure on source code produces a value of module complexity which characterizes a degree of structured-ness, i.e., how well the module conforms to precepts of structured programming. This is related to the Schutt, Gileadi, et. al. approach which deals in information entropy and software work. Using agglomerative clustering schemes, each node or additional nodes can be associated with a probability of misclassification for each existing cluster which can be used to determine some facet of modularity, organization, design quality, or sensitivity to change – obviously highly interpretive.

We see then, that with the various measures of design that can be used, each has a different shade of "quality" associated with it. Ideally, we would like to have a separate measure for each clearly identifiable software quality characteristic. To achieve this end, much research and experimentation is needed to calibrate and validate specific techniques to insure that the measures produced accurately reflect the attributes of the design problem.

APPENDIX

Measurements of a Requirements Expression

1. Completeness – The ultimate goal of this measure is to provide the analyst some means of determining whether or not more work needs to be done to a requirements expression to make it usable in determining a design expression, i.e., are the requirements that I presently have available of sufficient information content that a meaningful design activity can commence? A measure for completeness was derived by using two MEDL-R constructs and applied to sample "problem" sets of requirements. The measure is a percentage value which shows the degree to which all requirements are either RESOLVED in an appropriate manner or DERIVE others. (A greater percentage implies more completeness.) Appropriate resolution is determined by checking NATURE keywords against the type of RESOLUTION; e.g., NATURE: DATA implies that a DATA-RESOLUTION descriptor should be given

Sample Data:

Version	Pop.n	Derives	Resolved	%
I – Baseline	100	11	31	42
II – Baseline	7	1	0	14
II – A	12	1	0	8
II – B	15	1	5	40

The reduction in completeness in Version II-A from its Baseline is due to more requirements being added to the A revision. In Version II-B, further information was imparted to the requirements set by providing additional resolution of existing requirements.

2, Consistency Measurement – This measure should provide the analyst a means of determining the extent to which requirements in a set are isolated from one another. The term "isolation" means that property of a requirement that determines the extent to which it is bound to the total requirements set. The consistency measure used represents an average "strength" value, i.e., a large value implies a large degree of dependence among requirements. The value is determined by counting the number of names the requirements have in common.

Sample Data:

Version	Pop.n	Names	Strength
I – Baseline	100	47	.47
II – Baseline	7	4	.57
II – A	12	7	.68
II – B	15	9	.60

3. Complexity Measurement – This measure should provide a means to determine the inherent complexity of a set of requirements where the term "complexity" is still vague. However, it is clear that the greater the complexity of a set of requirements, the more difficult those requirements are to understand and hence more difficult to verify that they have been satisfied. So we chose a method that produces a probability that a node in a DERIVES tree is a non-terminal node. The higher this probability, the more complex the requirements expression.

Sample data:

Version	Pop.n	Non-Terms.	Probability
I – Baseline	100	27	.73
II – Baseline	7	6	.14
II – A	12	11	.08
II – B	15	13	.13

REFERENCES

1. Andreu, R. C., "A Systematic Approach to the Design of Complex Systems: An Application to DBMS Design and Evaluation," Center for Information Systems Research, Report #32. MIT, 1977.

2. Andreu, R. C., "Set Decomposition: Cluster Analysis and Graph Decomposition Techniques," CISR Preliminary Report, MIT/Sloan School, June 1977.

3. Gileadi, A. N. and Ledgard, H. F., "On a Proposed Measure of Program Structure," SIGPLAN Notices; May 1974.

4. Halstead, M., Elements of Software Science, Elsevier 1977.

5. Hamilton, M. and Zeldin, S., "HOS – A Methodology for Defining Software," IEEE Transactions, SE-2, March 1976.

6. Hartigan, J., Clustering Algorithms, Wiley, 1975.

7. Jackson, M., Principles of Program Design, Academic Press, 1975.

8. McCabe, T. J., "A Complexity Measure," IEEE Trans SE-2 No. 4, December 1976.

9. Myers, G. J., "An Extension to the Cyclomatic Measure of Program Complexity," SIGPLAN Notices, October 1977.

10. Myers, G. J., Reliable Software Through Composite Design, Petrocelli/Charter, New York, NY, 1975.

11. Paige, M. R., "On Partitioning Program Graphs, " IEEE Transactions, SE-3, No. 6, November 1977, p. 386.

12. Parnas, D. L., "On the Criteria to be Used in Decomposing Systems into Modules," CACM 15, 12 (December 1972), p. 1053.

13. Robinson, L., "The Relationship of System Families to HDM" (Hierarchical Development Methodology) – Stanford Research Institute, TR:CSL-50, June 1977.

14. Roubine, O., "The Design and Use of Specification Languages," SRI Tech Report CSL-48, October 1976 (AD/A 038-783).

15. Scheffer, P. A., "Computer-Aided Software Design," Martin Marietta Internal Report D-22R, December 1977.

16. Scheffer, P. A., and Velez, C. E. "On the Problem of Software Design and Measuring Quality," Proceedings NAECON, May 1978, p. 223.

REFERENCES

17. Schütt, D., "On a Hypergraph Oriented Measure for Applied Computer Science," CompCon Proceedings, September 1977, p. 295.

18. Silver, A. N., "On the Structural Decomposition and Heirarchical Recombination of Non-Directed Linear Graphs ...," Carnegie-Mellon Symposium on Constructive Approaches to Mathematical Models, July 1978.

19. Silver, A. N., "Structural Decomposition using Entropy Metrics", Proceedings of the 1978 conference on Information Sciences and Systems, JHU March 1978.

20. Stevens, W. P., Myers, G. J., and Constantine, L. L. Structural Design, Yourdan, Inc., 1975.

21. Teichroew, D. and Hershey, E. A. III, "PSL/PSA: A Computer-Aided Technique for Structured Documentation and Analysis of Information Processing Systems," IEEE Transactions on Software Engineering, SE-3/1, January 1977.

22. TRW, "Software Requirements Engineering Methodology," Report 27332-6291-024, September 1976; Ballistic Missile Defense Advanced Technology Center, Contract DASG60-75-C-0022.

23. Yourdan, E. and Constantine, L. L., Structured Design, Yourdan, Inc., 1975.

OMIT TO R107

PANEL #4

CURRENT ACTIVITIES AND FUTURE DIRECTIONS

Chairperson ..... Frank McGarry (GSFC)

Member #1 ..... Lorraine Duvall (IITRI)

Member # 2 ..... Vic Basili (University of Maryland)

Member # 3 ..... Chuck Everhart (Teledyne Brown Engineering)

Page intentionally left blank

Page intentionally left blank

THE DATA AND ANALYSIS CENTER FOR SOFTWARE

Lorraine M. Duvall
ITT Research Institute

The Data and Analysis for Software (DACS) is being established at the Rome Air Development Center (RADC) to serve as a centralized source for current, readily usable data and information concerning software technology. The major functions of this Center are to:

Develop and maintain a computer database of software data.

Establish a STINFO Library containing technical reports and a lessons-learned file.

Identify data requirements for research efforts and present as a guideline for collecting data.

Analyze the data in the database and the STINFO and produce data and information reports.

Establish a current awareness program including the publication of newsletters and involvement in technical meetings to disseminate information.

Produce and distribute data subsets, data compendiums, state-of-the-art reports, and bibliographies.

Provide consultation and inquiry services.

A functional model of DACS is presented in Figure 1. The functional flow consists of inputs from external sources, internal functions and operations, and outputs in the form of products and services. Two types of inputs are indicated; the first type is a material input, i.e. production/development data, textual documents, development reports, etc. The second type of input is a request for products or services, i.e. request for consulting services, data subsets, bibliographic searches, etc.

The outputs of DACS will be in the form of products and services. Typical products will include:

Data subsets

Data compendiums

Analysis reports

Bibliographies

Newsletters

Technical monographs

Services provided by DACS will generally include:

Bibliographic requests including the use of the STINFO Library

Answering inquiries on data compendium and/or dataset usage

Consultation on software technology

Participation in software technology committees and symposia

A computer database of software production/development data is being developed. The goal is to develop a database containing software environment, technology, resource utilization, production, and software characteristics data for reporting periods within the various phases of the software life cycle. The data may have been generated either manually or automatically through program support library systems, management control systems, and/or computer accounting systems. The data may also be submitted in the form of project schedule reports, problem reports/correction reports, design change requests, configuration management reports, and/or data contained within technical reports. The data within the database will be used to support research projects including estimating and monitoring the cost of producing computer software, measuring and predicting software reliability, error analysis, and productivity and complexity studies.

For more information on the Center and/or to receive the monthly newsletter, contact:

Lorraine M. Duvall
IIT Research Institute
Box 1355, Branch P.O.
Rome, New York 13440
Phone: 315 336-0937

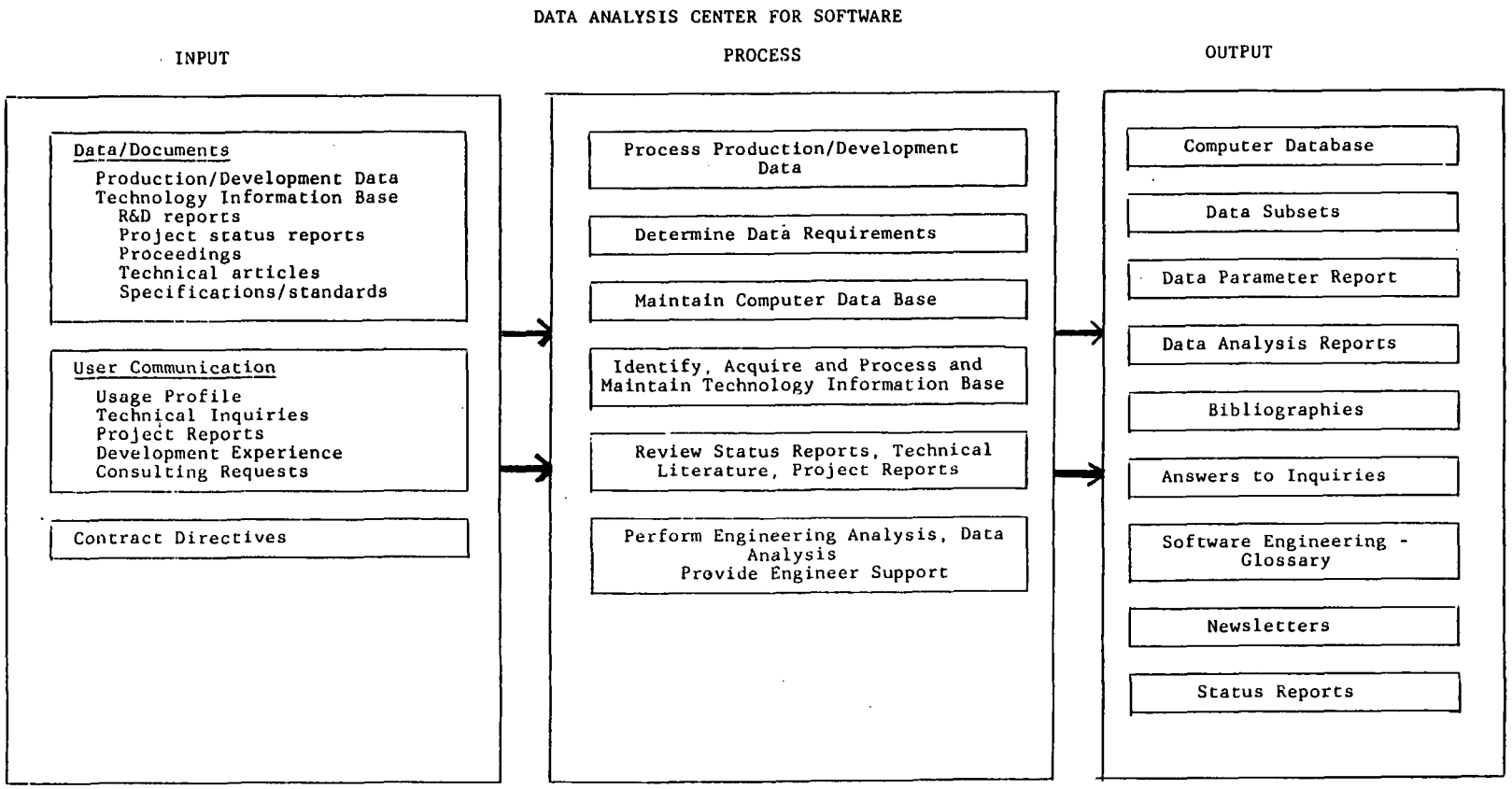


FIGURE -1 FUNCTIONAL FLOW

Page intentionally left blank

Page intentionally left blank

THE SOFTWARE ENGINEERING LABORATORY–1978*

Victor R. Basili and Marvin V. Zelkowitz
Department of Computer Science
University of Maryland
College Park, Maryland 20742

The Software Engineering Laboratory was started in August of 1976 as a joint venture between NASA/Goddard Space Flight Center and the Department of Computer Science of the University of Maryland. The purpose of the laboratory is to study the development of production software at NASA/GSFC and to recommend better ways to produce more reliable and less expensive products.

In the two years since the laboratory was organized, the empirical study of methodologies has become recognized as an important research topic. While the answers are not as yet known, we finally do have a handle on what questions to ask. Based upon a recent workshop [SLMW], the following are some of the important questions which must be resolved:

1. What are the issues to be studied? What are the various terms and do we have a consistent and complete set in order to define concepts like software, life cycle, requirements, specifications, etc.?

2. How do we classify the raw data that is collected? How are errors corrected? What personnel (programmers, management, support staff) are involved? How do different organizational and environmental considerations affect this?

3. How is consistent data collected by different organizations so that results can be compared?

4. What are the various models of software development and how are they validated with respect to this data?

5. How are the models refined to reflect local environmental changes?

These general questions are being interpreted in the NASA/GSFC environment to reflect that data we are collecting. In the Systems Development Section of NASA/GSFC, data is collected via a set of forms that are filled out by contract programmers during various stages of a project's development. The type of data collected includes:

1. Various environmental parameters (e.g., machine, storage and timing requirements, programming languages, etc.).

2. Manpower and computer usage on a weekly basis. This includes data on hours spent on various components of a system and what tasks were applied to those components (e.g., design, code, test).

*Research supported in part by Grant NSG–5123 from NASA/Goddard Space Flight Center to the University of Maryland.

3. Ratio of management to support staff to programmer effort.

4. Source code measures (e.g, number of modules, size, lines of code per hour, etc.).

5. Programmer overhead (e.g., travel, training, other activities).

6. Techniques used and their effectiveness (e.g., code reading, walkthroughs, design language, etc.).

The raw data is now being used in three distinct studies: (a) resource estimation, (b) product measures, and (c) error analysis.

(a) Resource Estimation

The resource estimates are based upon the earlier work of Putnam and Norden [Putnam]. For large scale projects (greater than 50 man years of effort), the Rayleigh curve has been found to fit manpower expenditures. The curve ($y = 2 K a t \exp(-a t^2)$) differs significantly from the "assumed" rectangle during the development cycle (e.g., 25 programmers for 2 years is 50 man years).

Within the NASA/GSFC environment, it has been found that the Rayleigh curve does not fit as well as it does with larger projects [Basili & Zelkowitz]. The probable reason is that these 6 to 12 man-year efforts have different characteristics than in larger projects. For example, with data being collected on a week-by-week basis, local perturbations (e.g., annual leave, sickness, computer down time) become much more pronounced. In addition, with only 1 year to develop a project, the effect of delay time makes needed organizational changes become more important than 5-or 6-year efforts.

Currently, work is progressing on evaluating alternatives to the Rayleigh curve model. The goal is to build an on-line program to feed back future estimates based upon initial data collected on any project.

(b) Product Measures

There are various measures of the "goodness" of a software product contained in the literature (Halstead, McCabe). The Laboratory is applying several of these measures as well as some new ones [Mills] and evaluating their effectiveness with respect to other parameters, such as error prediction and subjective judgments about quality. On an experimental data base, of multiple implementations of the same project under varying conditions, the concepts of modern programming practices (e.g., top down design, chief programmer teams, code reading) apparently improve software quality as quantitized by these measures. Work is being done to transfer the effective measures to the Laboratory environment.

(c) Error Analysis

Work is progressing in the classification and life-time of errors. A taxonomy of errors is being developed and metrics to measure reliability based upon found errors are being developed.

SUMMARY

In summing up these first two years of the laboratory, we have found that:

1. The process is slow and more complex than first imagined. We are forced to deal with practical realities in trying to apply the data to theoretical models.

2. We are lucky to be working within an understanding and intelligent environment. We would like to thank NASA/Goddard Space Flight Center and Computer Sciences Corporation for their support in this effort.

3. Progress is being made in

   a. resource estimation;

   b. basic data collection and the inter-relationships among the basic data;

   c. error analysis;

   d. product measures;

   e. automation of the process.

REFERENCES

[Basili and Zelkowitz]
V. Basili and M. Zelkowitz, Analyzing Medium Scale Software Development, Third International Conference on Software Engineering, Atlanta, Georgia, May 1978, pp. 116-123

[Halstead]
M. Halstead, Elements of Software Science, American Elsevier, 1977

[McCabe]
T. J. McCabe, A Complexity Measure, Transactions on Software Engineering 2, No. 4, 1976, pp. 308-320

[Mills]
H. Mills, Software Development, IEEE Transactions on Software Engineering 2, No. 4, 1976, pp. 265-273

[Putnam]
L. Putnam, A Macro Estimating Methodology for Software Development, IEEE Compcon, Washington, D.C., September 1976, pp. 138-143

[SLMW]
Proceedings of the Second Software Life Cycle Management Workshop, August 1978, IEEE Society Publication

Page intentionally left blank

Page intentionally left blank

SYSTEM REQUIREMENTS LANGUAGE FOUNDATION FOR SOFTWARE ENGINEERING

Charles R. Everhart
Teledyne Brown Engineering
Huntsville, Alabama

ABSTRACT

"System Requirements Language" here refers to those languages (defined by a formal set of syntax rules and semantics) whose purpose is the explicit and comprehensive expression of system definition and design facts. Not only is a formal requirements language necessary in specifying precisely the functional and performance characteristics of the system at all levels of definition and design, but at the same time this information can be used to predict the costs in time and money required to develop, implement, operate, and maintain a proposed system. Other benefits derived from this information include the direct generation of system and environment models used in the analysis of design solutions, direct generation of test criteria to be used during the test and integration phases of system development and the providing of a vehicle for maintaining configuration control throughout the life cycle of the system. "System" refers to not only software systems developed with "Software Development Methodologies", it also refers to the methodologies themselves.

1. MOTIVATION

According to a number of articles written recently [4,6] the costs of software development are becoming dominant (i.e., 50 to 90 percent of the cost of future data processing systems). In spite of the proliferation of programming languages [1] and software development techniques devised during the past 15 to 20 years, progress in reducing the costs of software has been disappointing. One source has indicated a decrease in productivity over this same period [3].

After a cursory analysis of major software developments, it appears that the industry has been attacking the wrong problem. A very large percentage of software development costs are associated with the definition, design, testing, and integration activities. The development of programming languages and techniques has been directed mainly toward the coding activity which represents only a small percentage of software development (as low as 17% for a large 7.5 year project [6]).

If the software industry is expecting to see large reductions in the cost of software, it must attack those problems connected with definition, design, testing and integration activities representing anywhere from 45 to 85 percent of most data processing development costs.

The first step in attacking the cost of these activities appears to be the development of a precise, yet convenient, system requirements specification and analysis language. Teledyne Brown Engineering has been actively involved in this type of development since 1971 and has devised a requirements language called IORL (Input/Output Requirements Language) which claims the objectives of the preceding discussion.

2. IORL

IORL is a formally defined language (syntactically and semantically [2,5] which uses a combination of both graphic symbols and mathematical notation to express system definition and design ideas. Block diagrams (analogous to those used in control theory) organized in a hierarchical manner identify the parts of a system and the interfaces between these parts at all levels of system definition and design.

Descriptions of each interface identified are contained in a set of tables called IOPT's (Input/Output Parameter Tables). Another diagram called in "IORTD" (Input/Output Relationships and Timing Diagram) is used to define the total transformation function from input to output as well as the response time requirements for each and every block in the hierarchy. These diagrams (analogous to a "Transfer Function" in control theory) provide the symbols for specifying the sequential, simultaneous, logical, mathematical and time requirements between inputs and outputs of each given block. The elements of IORL and hierarchical structure of requirements information are characterized in Figures 1 and 2.

3. IORL STORAGE AND RETRIEVAL FACILITY

Storage, retrieval and modification of IORL diagrams and tables has been implemented on a stand-alone PDP/11 based graphics terminal (GT44 and GT46) with 16K memory. The interactive graphics system includes a 17-inch refresh type graphic screen with lightpen capability as well as an electrostatic printer plotter which produces 8.5 x 11 inch copies of the screen. In edit mode, IORL information is entered by pointing the lightpen at a location on the screen and then pressing the keyboard button associated with the desired symbol. In display mode, system details are accessed by directing the lightpen to points of interest on the higher level diagrams of the hierarchy. This results in the subsequent display of these details on the screen. Requirements diagrams are stored on and retrieved from disk packs which are also a part of the facility.

4. BALLISTIC MISSILE DEFENSE (BMD) PARTITIONING STUDY EXAMPLE

Not only is a formal requirements language necessary in specifying precisely the functional and performance characteristics of the system at all levels of definition and design, but at the same time this information can be used to predict the costs in time and money required to develop, implement, operate and maintain a proposed system. Other benefits derived from this information include the direct generation of system and environment models used in the analysis of design solutions, direct generation of test criteria to be used during the test and integration phases of system development and the providing of a vehicle for maintaining configuration control throughout the life cycle of the system.

In an attempt to determine the feasibility of the preceding thesis, a study entitled "BMD Partitioning Study", was performed. The objective of this study was to demonstrate that certain quantitatiave characteristics, related to system development and operation costs, could be derived directly and mechanically from formally defined system requirements specifications. IORL was used as the language for specifying the definition and design requirements.

The demonstration consisted of four basic steps. First, the BMD system requirements and its environment were defined using the complete set of IORL symbols, tables and diagrams. This information, which represented the first level in the system hierarchy, was the only infor-

mation used in the subsequent requirements analysis and design activities. After checking the first level specification for completeness and consistency, the second step involved designing two different solutions to the BMD requirements, again expressing these solutions in IORL and placing this information in the second level of the system hierarchy. The purpose in specifying two design solutions was to compare the total system costs which resulted from each of the solutions. In the third step, each completely specified solution was validated against the BMD requirements by exercising the environment, BMD, and solution models (all written in IORL) and then comparing responses at the BMD/environment interfaces.

In this manner it was established that each solution responded to the environment exactly as required by the BMD requirements (model). If not, the design solution was corrected. In the final step, each solution was evaluated to determine its effect on total system costs. This evaluation, which was a combination of static and dynamic analysis of only that information contained in the IORL specifications, produced the summary partition evaluation results shown in Figure 6. These summary results were derived from a series of intermediate results which plotted for example bandwidth for each interface as a function of time, storage required as a function of time, functional speed required, etc.

Our experience with this study has resulted in the following conclusions:

- Quantitative measures related to the costs of a system can be determined from an analysis of system definition and design requirements.
- The requirements language used to specify system definition and design facts is the key factor in the success of the preceding demonstration (i.e., the language must have certain characteristics and enforce certain disciplines).
- The proper specification of definition and design requirements can provide information necessary to all phases of a system development (definition, design, implementation, test and integration).

5. FUTURE R&D ACTIVITIES

Teledyne Brown Engineering has plans to continue the development of IORL related techniques and tools. The immediate future calls for the development of the following computer utility packages:

- An extensive IORL diagnostic package (syntax analyzer)
- A set of configuration management tools
- An expended set of graphic editing features for the storage and retrieval system
- A utility program library
- A set of functional and analytic simulation compilers which will transform IORL information into FORTRAN or PASCAL simulation statements.

We have also experimented with the direct generation of assembler source code (Marco-II Assembler) from IORL information and have determined that the information content of IORL will support the development of a set of IORL compilers. The major problems associated with this last activity are the implementation of mathematical functions so easily represented in IORL.

REFERENCES

1. Shea, William E., "DOD-1, A Common Language", ISRAD in Touch, Volume 2, No. 1, February 1978.

2. Everhart, C. R., "IORL Analysts Users' Manual", Section 1 – Syntax, Teledyne Brown Engineering, May 1977.

3. Dolotta, T. A., et al, "Data Processing in 1980-1985", John Wiley and Sons, 1976.

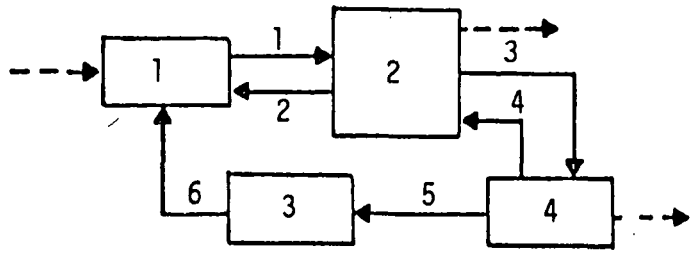
4. Boehm, B. W., "Software and Its Impact: A quantitative Assessment", Datamation, pp. 48-59, May 1973.

5. Everhart, C. R., "IORL Analysts Users' Manual", Section 2 – Semantics, Teledyne Brown Engineering, May 1977.

6. Ramamoorthy, C. V., et al., "Software Requirements and Specifications: Status and Perspectives", Engineering Research Recommendations, Draft Appendix A, August 12, 1977.

IORL ELEMENTS

1



SBD

2

IOPT 1

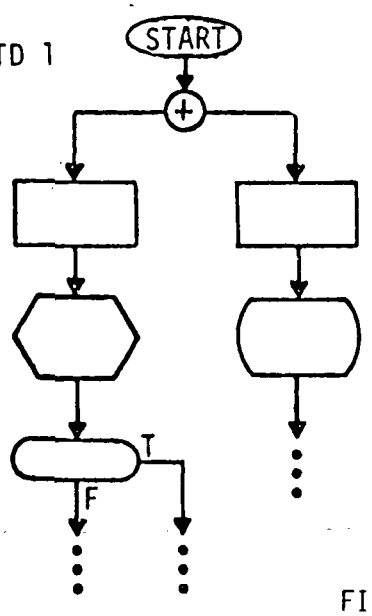
G	PARAMETER	VALUES	UNITS

IOPT 6

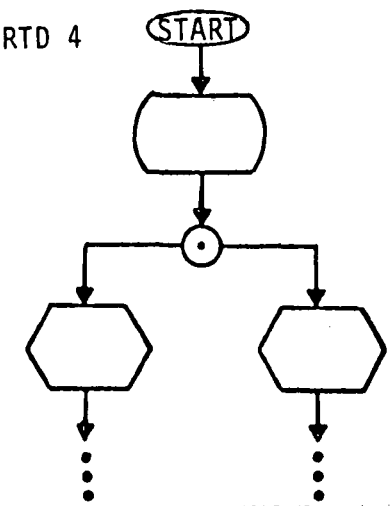
IOPT's

3

IORTD 1



IORTD 4



IORTD's

FIGURE 1. IORL ELEMENTS

C. Everhart
Teledyne Brown Eng.

IORL INFORMATION HIERARCHY DIAGRAM

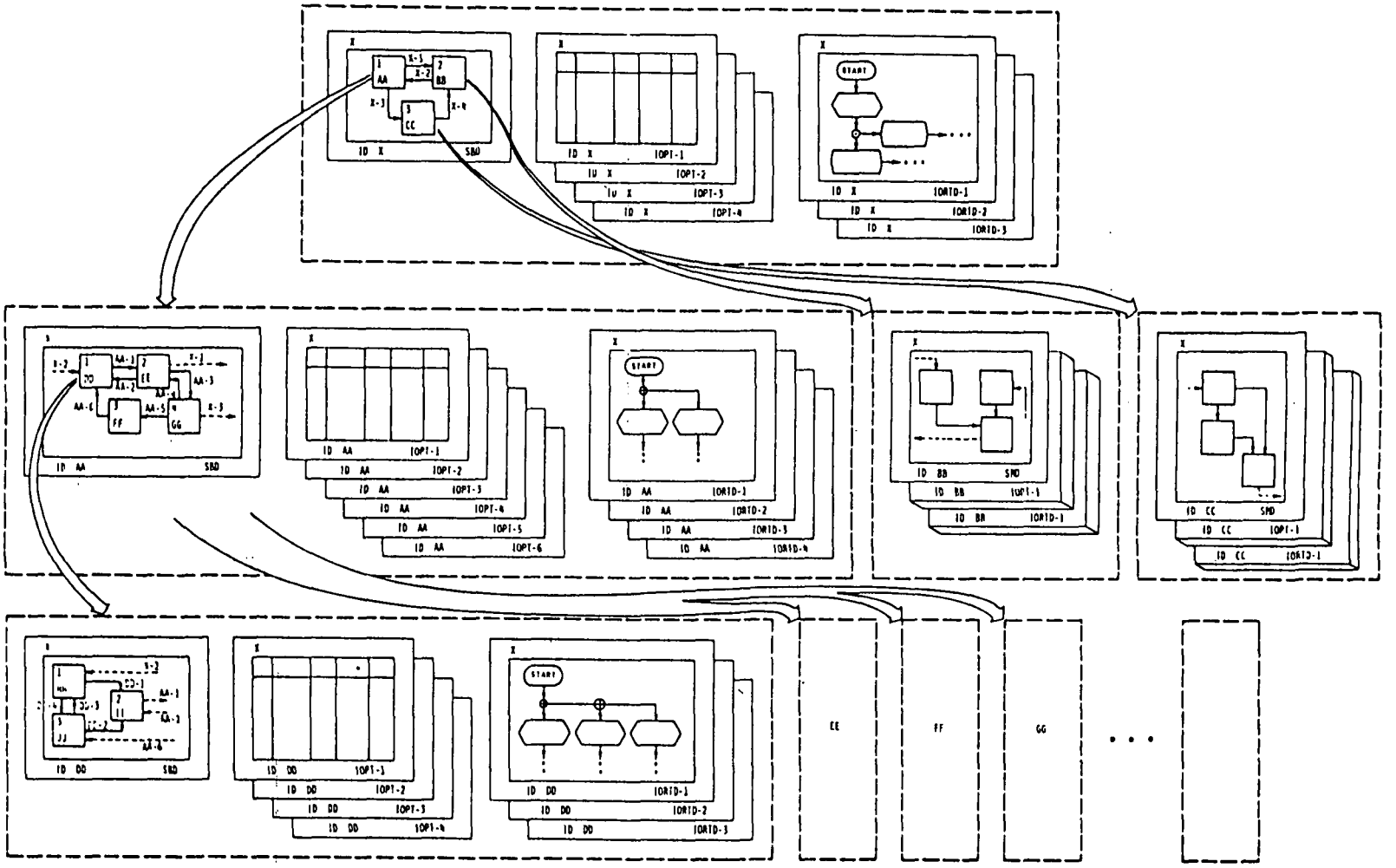


FIGURE 2. IORL INFORMATION HIERARCHY DIAGRAM

C. Everhart
Teledyne Brown Eng.

CURRENT CAPABILITIES – STORAGE AND RETRIEVAL
OF REQUIREMENTS USING INTERACTIVE GRAPHICS

- BASIC STORAGE AND RETRIEVAL OF INFORMATION
  - ▲ DISK PACKS (1200 PAGES/PACK)
  - ▲ CLASSIFIED FACILITY
- ON LINE EDITING
  - ▲ CRT
  - ▲ LIGHT PEN
  - ▲ FUNCTION KEYS
- HARDCOPY
  - ▲ REPORT QUALITY
- DOCUMENTATION AUDIT
- MERGE SYSTEM PAGES
- AUTOMATIC ACCOUNTING

Figure 3. Current Capabilities – Storage and Retrieval of Requirements Using Interactive Graphics

C. Everhart
Teledyne Brown Eng.

SYSTEM CONFIGURATION

- PDP-11/40 WITH 16K MEMORY
- 17" GRAPHIC DISPLAY WITH LIGHTPEN
- TELETYPE
- 2 DISK DRIVES
- ELECTROSTATIC PRINTER/PLOTTER (8½ X 11 FAN FOLD)

Figure 4. System Configuration

C. Everhart
Teledyne Brown Eng.

COMPUTERIZED ANALYSIS (PARTITIONING STUDY)

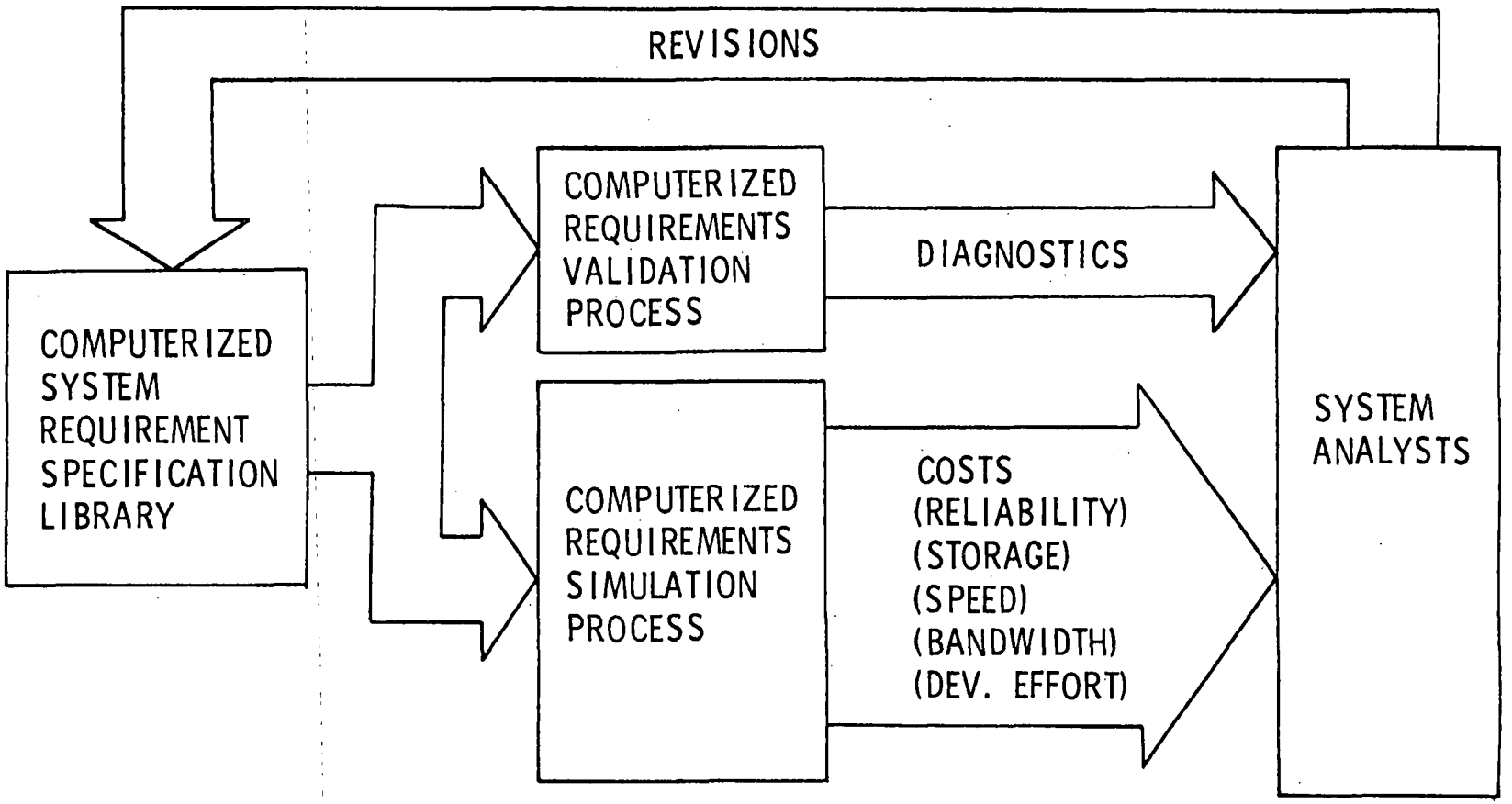


FIGURE 5. COMPUTERIZED ANALYSIS (PARTITIONING SYSTEM)

C. Everhart
Teledyne Brown Eng.

PARITITION EVALUATION RESULTS

	PARTITION 1				PARTITION 2			
	GND	TRKER	SENSOR	TOTAL OR WORST	GNDSYS	STTCOR	SENS	TOTAL OR WORST
RELIABILITY	0.975	0.900	0.975	0.941	0.980	0.980	0.980	0.941
STORAGE (KBITS)	23.5	389.3 (2)	2.67	804.8	138.6	284.4	27.9	432.7
PROCESS SPEED (η SEC)	5.44	5.85	5.56	5.44	0.814	0.814	0.814	0.814
PROGRAMMING COMPLEXITY (MAN-MONTHS)				35.65				26.3
PEAK BANDWIDTH (MBITS/SEC)				2.52				20.94

FIGURE 6. PARTITION EVALUATION RESULTS

C. Everhart
Teledyne Brown Eng.

C. Everhart
Teledyne Brown Eng.
Page 12 of 13

TOTAL SYSTEM DEVELOPMENT PROCESS

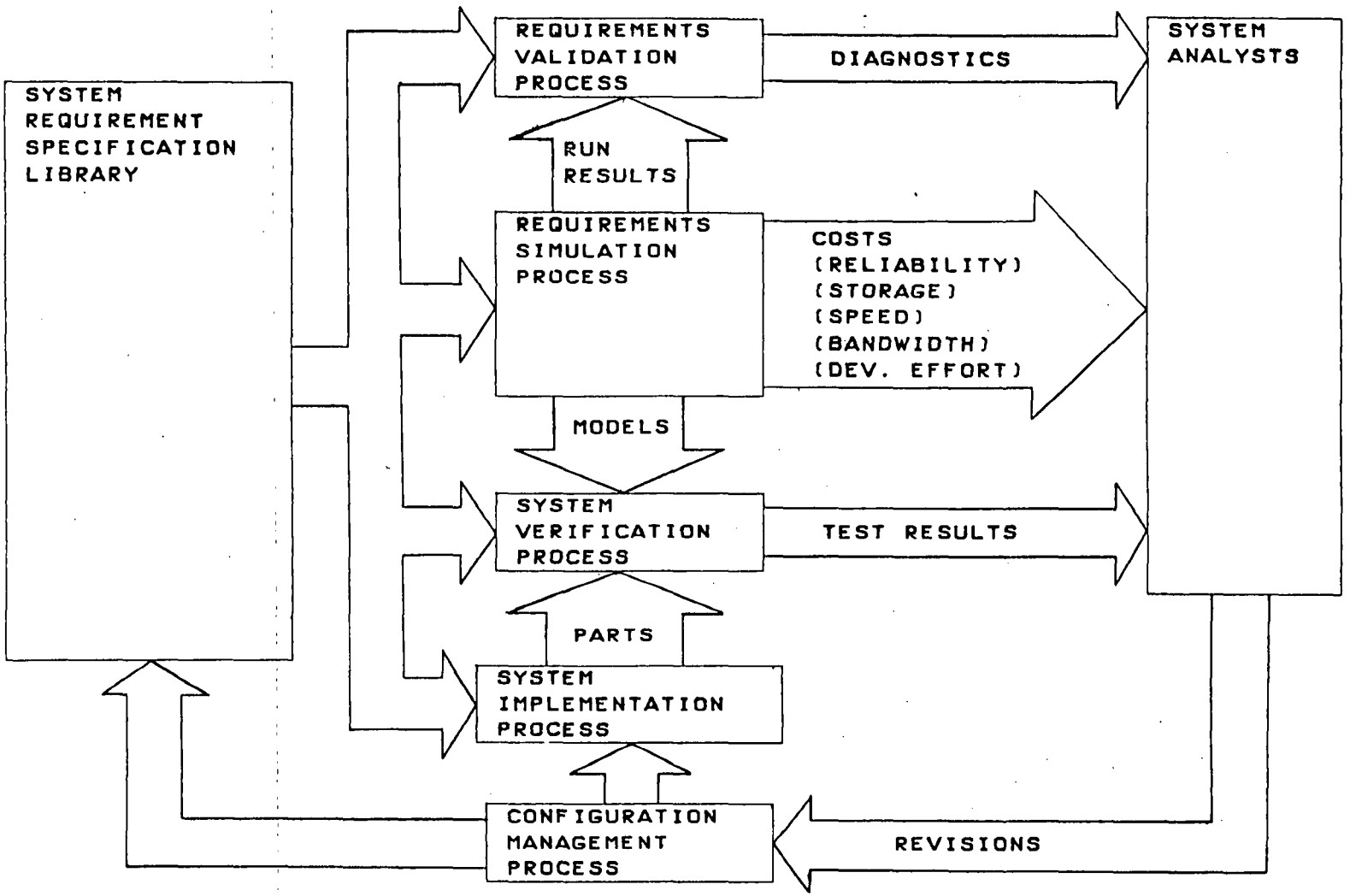


FIGURE 7. TOTAL SYSTEM DEVELOPMENT PROCESS

FUTURE IORL RESEARCH AND DEVELOPMENT

- DIAGNOSTICS PACKAGE (SYNTAX ANALYZER)
- CONFIGURATION MANAGEMENT TOOLS
- EXPANDED EDITING FEATURES (GRAPHICS)
- UTILITY PROGRAM LIBRARY (STATIC ANALYSIS)
- SIMULATION COMPILER
  - ▲ FUNCTIONAL: "IORL" TO "MODELER" TRANSLATOR
  - ▲ ANALYTIC: "IORL" TO "FORTRAN" TRANSLATOR
- COMPILER
  - ▲ "IORL" TO "FORTRAN" TRANSLATOR
  - ▲ "IORL" TO "PDP-11" MACRO-ASSEMBLER" TRANSLATOR

Figure 8. Future IORL Research and Development

C. Everhart
Teledyne Brown Eng.

Page intentionally left blank

Page intentionally left blank

SOFTWARE ENGINEERING WORKSHOP ATTENDEES

B. Aygun
Data Processing Marketing Group
IBM
B/931 D/Z60
Box 390
Poughkeepie, NY 12602
(914-485-8445)

V. Basili
University of Maryland
Computer Sciences Dept.
College Park, MD 20740

J. Bishop
NASA Headquarters
Code TN-1
600 Independence Avenue
Washington, DC 20546
(755-2325)

D. Bond
CSTA-Dept. 590
Aerospace Building
10210 Greenbelt Road
Seabrook, MD 20804

D. Brooks
IBM
10215 Fernwood Road
Bethesda, MD 20034

R. Broskio
Computer Sciences Corporation
8728 Colesville Road
Silver Spring, MD 20910

V. Brown
NASA Goddard Space Flight Center
Code 582.1
Greenbelt, MD 20771

B. Chu
Computer Sciences Corporation
8728 Colesville Road
Silver Spring, MD 20910

J. Crowley
Code 582.1
NASA Goddard Space Flight Center
Greenbelt, MD 20910

B. Curtis
General Electric
1755 Jefferson Davis Highway
Suite 200
Arlington, VA 22202

S. DePriest
CSTA
NASA Goddard Space Flight Center
Code 582
Greenbelt, MD 20771
982-4725

B. DeWolf
Charles S. Draper Labs
Mail Station 64
555 Technology Square
Cambridge, MA 02139

R. Durachka
NASA Goddard Space Flight Center
Code 565.3
Greenbelt, MD 20910

L. Duvall
IIT Research Institute
P. O. Box 1355
Branch P.O. Rome, NY 13440

C. Everhart
Teledyne-Brown Engineering
Cummings Research Park, M.S. 202
Huntsville, AL 35807
(205-536-4455 Ext. 610)

C. Felix
IBM
10215 Fernwood Road
Bethesda, MD 20034

C. Goorevich
Computer Sciences Corporation
8728 Colesville Road
Silver Spring, MD 20910

J. Green
TRW Defense and Space Systems Group
Bldg. 65, Room 1638
1 Space Park
Redondo Beach, CA 90278
(213-535-4321)

J. Grondalski
Computer Sciences Corporation
8728 Colesville Road
Silver Spring, MD 20910

J. Harris
NASA Langley Research Center
Hampton, VA 23665

F. Harris
IBM
10215 Fernwood Road
Bethesda, MD 20034

B. Hodges
NASA Marshall Space Flight Center
Code AH31
Huntsville, AL 35812
(872-2121)

P. Hsia
Computer Sciences Dept.
University of Alabama
P.O. Box 1247
Huntsville, AL 35807
(895-6088)

A. James
8300 South Whitesburg Drive
Huntsville, AL 35802
(205-453-1389)

T. King
Computer Sciences Corporation
8728 Colesville Road
Silver Spring, MD 20910

J. Knight
Analysis and Computation Division
NASA Langley Research Center
Hampton, VA 23665

P. Knowles
CSTA
NASA Goddard Space Flight Center
Code 582
Greenbelt, MD 20771

T. Kurihara
2058 Carrhill Road
Vienna, VA 22180

K. S. Liu
Computer Sciences Corporation
8728 Colesville Road
Silver Spring, MD 20910

R. Luczak
Computer Sciences Corporation
8728 Colesville Road
Silver Spring, MD 20910

F. McGarry
Code 582.1
NASA Goddard Space Flight Center
Greenbelt, MD 20910

M. McKenzie
JPL
Mail Stop 264/801
4800 Oak Grove Drive
Pasadena, CA 91103

P. Milliman
G.E.
1755 Jefferson Davis Highway
Suite 200
Arlington, VA 22202

K. Moe
Code 514
NASA Goddard Space Flight Center
Greenbelt, MD 20771

G. Page
Computer Sciences Corporation
8728 Colesville Road
Silver Spring, MD 20910

J. Palaimo
Rome Air Development Center
Griffis Air Force Base
BR. Rome, NY 13441

F. Petry
University of Alabama
Huntsville, AL 35807

L. Putnam
Quantitative Software Management
1057 Waverly Way
McLean, VA 22101

G. Picasso
Computer Sciences Dept.
University of Maryland
College Park, MD 20740

R. Reiter
Computer Sciences Dept.
University of Maryland
College Park, Md 20740

C. Rumore
Computer Sciences Corporation
8728 Colesville Road
Silver Spring, MD 20910

P. Ryan
Science Application, Inc.
2109 W. Clinton Avenue
Suite 800
Huntsville, AL 35805
(205-533-5900)

P. Scheffer
Martin Marietta Corporation
Mail Stop 0422
P.O. Box 179
Denver, CO 80201

L. Schumacher
DOD/PESO
c/o DLA CAMERON STATION
Alexandria, VA 22314

E. Senn
Analysis and Computation Division
NASA Langley Research Center
Hampton, VA 23665

S. Sheppard
G.E.
1755 Jefferson Davis Highway
Suite 200
Arlington, VA 22202

J. Stephens
Computer Sciences Corporation
8728 Colesville Road
Silver Spring, MD 20910

D. Sung
G.E.
5070 Herzel Place
Beltsville, MD 20705

K.K. Tasaki
Code 582.1
NASA Goddard Space Flight Center
Greenbelt, MD 20771

W. Truszkowski
Code 514
NASA Goddard Space Flight Center
Greenbelt, MD 20910

C. E. Velez
Martin Marietta Corporation
P.O. Box 179
Denver, CO 80201

D. Weiss
Naval Research Lab
Washington, DC 20375

D. Wilson
Computer Sciences Corporation
8728 Colesville Road
Silver Spring, MD 20910

D. Wychoff
Computer Sciences Corporation
8728 Colesville Road
Silver Spring, MD 20910

M. Zelkowitz
Computer Sciences Dept.
University of Maryland
College Park, MD 20740