MODELS AND METRICS FOR SOFTWARE MANAGEMENT AND ENGINEERING

V. R. Basili
University of Maryland
College Park, Maryland

ABSTRACT

This paper attempts to characterize and present a state of the art view of several quantitative models and metrics of the software life cycle. These models and metrics can be used to aid in managing and engineering software projects. They deal with various aspects of the software process and product, including resource allocation and estimation, changes and errors, size, complexity and reliability. Some indication is given of the extent to which the various models have been used and the success they have achieved.

INTRODUCTION

The past few years have seen the emergence of a new quantitative approach to software management and software engineering. It includes the use of models and metrics based on historical data and experience. It covers resource estimation and planning, cost, personnel allocation, computer use, and quality assurance measures for size, structure and reliability of the product.

A quantitative methodology is clearly needed to aid in the software development process. It is needed for understanding and comparison. It was said by Lord Kelvin that if you cannot measure something, then you do not understand it. This is certainly true in the software development domain and is the reason why various models and metrics have been developed, tested, refined and established as aids. One needs models and quantification for comparisons. In cost tradeoffs, for example, it is important to know whether to add another feature, how much an extra level of reliability will cost, or whether a modification to an existing system will be cost effective.

It should be noted, however, that the quantitative approach should augment and not replace good management and engineering judgment. Models and metrics are only tools for the good manager and engineer. This is especially true since the state of the art is newly emerging and not yet well established. Some models and metrics have only been proposed but not fully tested. Others have been tested only in the environment in which they have been developed. However, more and more are being tested and used in environments other than that of the developer. In this paper, some indication of the level of experience with the models or metrics discussed will be given.

Models and metrics must be established via sound testing and experimentation and, before using a model, the manager or engineer should have sufficient knowledge about how much to trust the results of the model. This requires insight into the model, a known confidence level with regard to its reliability and, most important, knowledge of the activity being modeled. None of these models are black boxes and should not be treated as such. Thus, before applying any model, the user should know the nature of his project, whether the assumptions of the model match the environment of his project, and the weaknesses of the model so that he can be careful in evaluating the results.

In what follows, we will cover a large, though by no means exhaustive, set of models. The emphasis will be on those areas where quantitative management can give the greatest payoff. We will discuss process-oriented measures such as size, complexity, and reliability. Each of the measures will be treated to varying degrees. The emphasis will be on categorizing the measures, defining a typical measure or set in the category, and pointing out other measures only when they are different. The references in the back of the paper should help the interested reader pursue a particular measure further or find additional measures not mentioned in this paper.

PROCESS MEASURES

Resources

It is important that we have a better understanding of the software development process and be able to control the distribution of resources such as computer time, personnel, and dollars. We are also interested in the effect of various methodologies on the software development process and how they change the distribution of resources. For this reason, we are interested in knowing the ideal resource allocation, how it may be modified to fit the local environment, the effect of various tradeoffs, and what changes should be made in the methodology or environment to minimize resources expenditure.

There has been a fair amount of work towards developing different kinds of resource models. These models vary in what they provide (e.g., total cost, manning schedule) and what factors they use to calculate their estimates. They also vary with regard to the type of formula, parameters, use of previous data, and staffing considerations. In an attempt to characterize the models, we will define the following set of attribute pairs. Models can be characterized by the type of formula they use to calculate total effort. A single variable model uses one basic variable as a predictor of effort, while a multi-variable model uses several variables. A model may be static with regard to staffing, which means a constant formula is used to determine staffing levels for each activity, or it may be dynamic, implying staffing level is part of the effort formula itself. Within the static multi-variable models, there are various subcategories: adjusted baseline, adjusted table-driven, and multi-parameter equation. The adjusted baseline uses a single variable baseline equation which is adjusted in some way by a set of other variables. An adjusted table-driven model uses a baseline estimate which is adjusted by a set of variables where the relationships are defined in tables built from historical data. A multi-parameter model contains a base formula which uses several variables. A model may be based upon historical data or derived theoretically. An historical model uses data from previous projects to evaluate the current project and derive the weights and basic formula from analysis of that data. For a theoretical model, the formula is based upon assumptions about such things as how people solve problems. One last categorization is that some models are macro models, which means they are based upon a view of the big picture, while others are micro models in that the effort equation is derived from knowledge of small pieces of information scaled up. We will try to discuss at least one model in each of these categories.

Static single variable models. The most common approach to estimating effort is to make it a function of a single variable, project size (e.g., the number of source instructions or object instructions). The baseline effort equation is of the form

$$\text{EFFORT} = a * \text{SIZE}^b$$

where a and b are constants. The constants are determined by regression analysis applied to historical data. In an attempt to measure the rate of production of lines of code by project as influenced by a number of product conditions and requirements, Walston and Felix (1) at IBM Federal Systems Division started with this basic model on a data base of 60 projects of 4,000 to 467,000 source lines of code covering an effort of 12 to 11,758 man months. The basic relation they derived was

$$E = 5.2L^{.91}$$

where E is the total effort in man months and L is the size in thousands of lines of delivered source code, including comments. Beside this basic relationship, other relations were defined. These include the relationships between documentation DOC (in pages) and delivered source lines

$$DOC = 49L^{1.01}$$

project duration D (in calendar months) and lines of code

$$D = 4.1L^{.36}$$

project duration and effort

$$D = 2.47E^{.35}$$

and average staff size S (total staff months of effort/duration) and effort

$$S = .54E^{.6}$$

The constants a and b are not general constants. They are derived from the historical data of the organization (in this case, IBM Federal Systems Division). They are not necessarily transportable to another organization with a different environment. For example, the Software Engineering Laboratory (SEL) on a data base consisting of 15 projects of 1.5 to 112 thousand source lines of code covering efforts of 1.8 to 116 staff months have calculated for their environment the following set of equations (2):

$$E = 1.4L^{.94}$$

$$DOC = 29.5L^{.92}$$

$$D = 4.4L^{.267}$$

$$D = 4.4E^{.26}$$

$$S = 2.3E^{.74}$$

Some other variables, including different ways of counting code, were measured by the Software Engineering Laboratory and the equations derived are given here. Letting DL = number of developed, delivered lines of source code (new code + 20% of reused code), M = number of modules, DM = total number of developed modules (all new or more than 20% new) we have

$$E = 1.58DL^{.96}, \quad E = .063M^{1.186}, \quad E = .19DM^{1.0},$$

$$D = 4.6DL^{.28}, \quad D = 2.0M^{.33}, \quad D = 2.5DM^{.3},$$

$$D = 2.0D^{.26}, \quad DOC = 35.7DL^{.92}, \quad DOC = 1.5M^{1.17},$$

$$DOC = 4.8DM^{.99}$$

Most of the SEL equations lie within one standard error of the IBM equation and, since the SEL environment involves the development of more standardized software (software the organization has experience in building), the lower effort for more lines of code seems natural. It is also worth noting that the basic effort vs. lines-of-code equation is almost linear for the SEL--more linear than the Walston/Felix equation. Remember that the project sizes are in the lower range of the IBM data. Lawrence and Jeffery (3) have studied even smaller projects and discovered that their data fits a straight line quite well, i.e., their baseline effort equation is of the form

$$\text{EFFORT} = a * \text{SIZE} + b$$

where again a and b are constants derived from historical data. The implication here is that the equation becomes more linear as the project sizes decrease.

Static multi-variable models. Another approach to effort estimation is what we will call the static multi-variable model. A resource estimate here is multi-variable because it is based on several parameters, and static because a single effort value is calculated by the model formula. These models fall into several subcategories. Some start with the baseline equation just discussed based on historical data and adjust the initial estimate by a set of variables which attempt to incorporate the effects of important product and process attributes. In other models, the baseline equation itself involves more than one variable.

The models in the adjusted baseline class differ in the set of attributes that they consider important to their application area and development environment, the weights assigned to the attributes, and the constants of the baseline equations.

Walston and Felix (1) calculated a productivity index by choosing 29 variables that showed a significantly high correlation with productivity in their environment. It was suggested that these be used in estimating and were combined in a productivity index

$$I = \sum_i w_i \; x_i$$

where I is the productivity index, w_i is a factor weight based upon the productivity change for factor i and $x_i = +1, 0,$ or -1, depending on whether the factor indicates increased, nominal or decreased productivity.

One model that fits into the single-parameter baseline equation with a set of adjusted multipliers is the model of Boehm (4), whose baseline effort estimate relies only upon project size. His set of attributes are grouped under four areas: (1) product--required fault freedom, data base size, product complexity, adaptation from existing software; (2) computer--execution time constraint, machine storage constraint, virtual machine volatility, computer response time; (3) personnel--analyst capability, applications experience, programmer capability, virtual machine experience, programming language experience; (4) project--modern programming practices, use of software tools, required development schedule. For each attribute Boehm gives a set of ratings ranging from very low to very high and, for most of the attributes, a quantitative measure describing each rating. The ratings are meant to be as objective as possible (hence the quantitative definitions), so that the person who must assign the ratings will have some intuition as to why each attribute could have a significant effect on the total effort. In two of the cases where quantitative measures are not possible, required fault freedom and product complexity, Boehm provides a chart describing the effect on the development activities or the characteristics of the code corresponding to each rating. Associated with the ratings is a chart of multipliers ranging from about .1 to 1.8. Another model which falls into this category is the model of Doty (5). The Doty model, however, provides a different set of weights for different applications besides two ways to estimate size.

One model which falls into the category of adjusted table-driven is that of Wolverton (6). Here the basic algorithm involves categorizing the software routines. The categories include control, I/O, pre- or post-algorithm processor, algorithm, data management, and time critical routines. Each of these routines has its own cost-of-development curve, depending upon the degree of difficulty (easy, medium, or hard) and the newness of the application (new or old). The cost is then the number of instructions by category and degree of difficulty times the corresponding cost taken from a table. Another model of this type, but more simplistic, is Aron (7).

The GRC model (8) involves a set of equations derived from historical data and theory for the various activities, several of which are multi-parameter equations of more than one variable. For example, the equation for code development is

$$MM_{CD} = .9773 \times N_{OF}^{1.2583} \times e^{-.08953 * Y_{EXP}}$$

where MM_{CD} is the baseline staff months for code development task group for a subsystem, N_{OF} = the number of output formats for a subsystem and Y_{exp} is the average years of staff experience in code development. It is worth noting that size of the code is not a factor in this formula. Other formulas exist for the effort involved in analysis and design, system level testing, documentation, installation, training, project control, elapsed time and a reasonable check for the total staff months for the project (MM_{PROJ})

$$MM_{PROJ} = .0218 * ((2 + N_{OF}) * \ln(2 + N_{OF}))^{1.71}$$

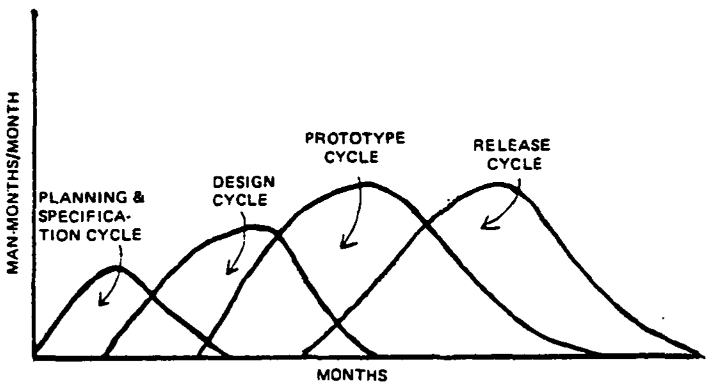
where N_{OF} is as defined above.

Dynamic multi-variable models. Once an effort estimate is made, the next question of concern is how to assign people to the project so that the deadlines for the various development activities will be met. Here again there are basically two approaches: the one empirical, the other theoretical. Each of the methods discussed so far uses the empirical approach which tries to identify the activities which are a part of the development process of a typical project for their software house. Then, using accounting data from past projects, they determine what percentage of the effort was expended on each activity. These percentages serve as a baseline and are intuitively adjusted to meet the expected demands of a new project. For example, in the Wolverton model, total cost is allocated into five major subareas: analysis cost (20% of total), design cost (18.7% of total), coding cost (21.7% of total), testing cost (28.3% of total) and documentation cost (11.3% of total). Each of these subarea costs are subdivided again, depending upon the activities in the subareas. In this way, each activity can be staffed according to its individual budget. Allocation of time is determined by history and good management intuition.

The theoretical approach attempts to justify its resource expenditure curve by deriving it from equations which model problem-solving behavior. In other words, the resource model lays out the staffing across time and within phases. We will refer to this approach as the dynamic multi-variable model. It is dynamic because the model produces a curve which describes the variation of staffing level across time. The model is multi-variable because it involves more than one parameter.

Two models in this category will be discussed which differ in the assumptions they make. The first model, which is the most widely known and used, is the Putnam model (9).

The model is based on a hardware development model (10) which noted that there are regular patterns of manpower buildup and phase-out independent of the type of work done. It is related to the way people solve problems. Thus, each activity could be plotted as a curve which grows and then shrinks with regard to staff effort across time. For example, the cycles in the life of a development engineering project look as follows:



Similar curves were derived by Putnam for software cycles which are: planning, design and implementation, testing and validation, extension, modification and maintenance.

The theoretical basis of the model is that software development is a problem-solving effort and design decision-making is the exhaustion process. The various development activities partition the problem space into subspaces corresponding to the various stages (cycles) in the life cycle. A set of assumptions is then made about the problem subset: (1) the number of problems to be solved is finite, (2) the problem-solving effort makes an impact on and defines an environment for the unsolved problem set, (3) a decision removes one unsolved problem from the set (assumes events are random and independent) and (4) the staff size is proportional to the number of problems "ripe" for solution. Because the model is theoretically based (rather than empirically based) some motivation for the equation is given. Consider a set of independent devices under test (unsolved problem set) subject to some environment (the problem-solving effort) which generates shocks (planning and design decisions). The shocks are destructive to the devices under test with some dependent conditional probability distribution p(t) which is random and independent with some rate parameter λ. Assume the distribution is Poisson and let T be a random variable associated with the time interval between shocks

$$\Pr(T > t) = \Pr \text{ (no event occurs in interval } (o, t)) \quad (1)$$

where t = o is the time of the most recent shock letting p(t) be the conditional probability of a failure given that a shock has occurred and λ be the Poisson rate parameter, then

$$\Pr(T > t) = e^{-\lambda \left(\int_0^t p(x)\,dx\right)} \quad (2)$$

and

$$\Pr(T \leq t) = 1 - e^{-\lambda \left(\int_0^t p(x)\,dx\right)} \quad (3)$$

and the p.d.f. associated with (3) is

$$f(t) = \lambda \cdot p(t) \cdot e^{-\lambda \left(\int_0^t p(x)\,dx\right)},\ t \geq o$$

This leads to the class of Weibull distributions (known in reliability work) with the physical interpretation that the probability of devices succumbing to destructive shocks is changing with time. Based upon observed data on engineering design projects, a special case of (3) can be used

$$y = f(t) = 1 - e^{-at^2} \quad (4)$$

$$\text{where} \quad p(t) = \alpha t \quad (5)$$

$$\text{and} \quad a = \frac{\lambda\alpha}{2} \quad (6)$$

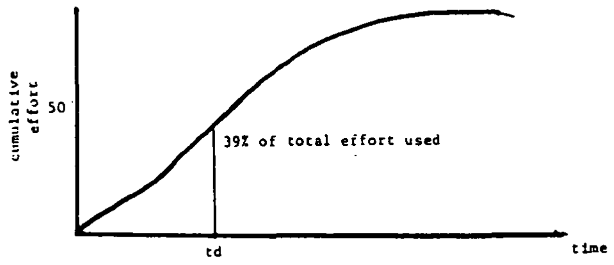
Note that this implies engineers learn to solve problems with an increasing effectiveness (i.e., familiarity with the problems at hand leads to greater insight and sureness). Parameter a consists of an insight generation rate λ and a solution finding factor α. Equation (5) is a special linear case of the family of learning curves: $y = a\,x^b$.

Equation (4) is then the normalized form of the life cycle equation. By introducing a parameter (K) xpressed in terms of effort, we get an effort curve, the integral form of the life cycle equation

$$y = K * (1 - e^{-at^2})$$

where

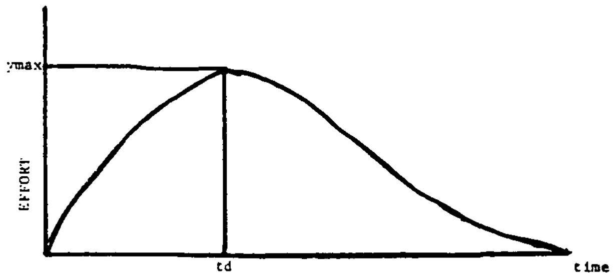
y	is the cumulative manpower used through time t
K	is the total manpower required by the cycle stated in quantities related to the time period used as a base, e.g., man-months/month
a	is a parameter determined by the time period in which y′ reaches its maximum value (shape parameter)
t	is time in equal units counted from the start of the cycle



The life cycle equation (derivative form) is

$$y' = 2\,K\,a\,t\,e^{-a\,t^2}$$

where y′ is the manpower required in time period t stated in quantities related to the time period used as a base and K is the total manpower required by the cycle stated in the same units as y′.



The curve (called the Rayleigh Curve) represents the manpower buildup. The sum of the individual cycle curves results in a pure Rayleigh shape. Software development is implemented as a functionally homogenous effort (single purpose). The shape parameter a depends upon the point in time at which y′ reaches its maximum, i.e.

$$a = {}^{1}/_{2t_d{}^2}$$

where t_d is the time to reach peak effort. Putnam has empirically shown t_d corresponds closely to the design time (time to reach initial operational capability). Substituting for a we can rewrite the life cycle equation as

$$y' = \frac{K}{t_d{}^2} * t e^{-t^2/2\,t_d{}^2}$$

The equations given are for the entire life cycle. To find development effort only

take

$$y = K * (1-e^{-at^2})$$

substitute $a = {}^{1}/_{2t_d^2}$

$$y = K * (1-e^{(-t^2/2t_d^2)})$$

then the development effort is time to t_d

$$y = K * (1-e^{(-t^2/2t_d^2)})$$

$$= K * (1-e^{-.5})$$

$$= .3935K$$

or DE = 40% of LC effort

The life cycle and development costs may be calculated by multiplying the cost for that cycle by staff year cost

$LC = K*MC

where MC = mean cost (in $) per man year of effort

K = total manpower (in man years) used by the project

(Note: the equation neglects computer time, inflation, overtime, etc.)

and

$DEV = MC * (.3935K) ≈ .4 * $LC

Putnam found that the ratio $K/(t_d^2)$ has an interesting property. It represents the difficulty of a system in terms of programming effort required to produce it. He defines

$$D = K/(t_d^2)$$

To illustrate how management decisions can influence the difficulty of a project, assume a system size of K = 400 MY and t_d = 3 years. Then the difficulty D = 400 / 9 = 44.4 man years per year squared.

Consider a management decision to cut the life cycle cost of the system by 10%. Now, K = .9 * (400) = 360 MY and D = 360 / 9 = 40. This results in a 10% decrease in assumed difficulty of the project. This decision assumes the difficulty is less than it really is, and the result is less product.

Now consider the more common case of attempted time compression. Assume management makes a decision to limit the expended effort to 400 MY, but wants the system in 2.5 years instead of 3 years. Now, K = 400 MY, t_d = 2.5 years, and D = 400 / 6.25 = 64 (a 44% increase). The result of shortening the natural development time is a dramatic increase in the system difficulty.

The Putnam model generates some interesting notions. Productivity is related to the difficulty and the state of technology; management cannot arbitrarily increase productivity nor can it reduce development time without increasing difficulty. The tradeoff law shows the cost of trading time for people.

In deriving an alternate model, Parr (11) questions the assumption of the Rayleigh equation that the initially rising work rate is due to the linear learning curve which governs the skill available for solving problems. He argues that the skill available on a project depends on the resources applied to it and that the assumption confuses the intrinsic constraints on the rate at which software can be developed with management's economically-governed choices about how to respond to these constraints.

As an alternative to this assumption, his model suggests that the initial rate of solving problems is governed by how the problems in the project are related, i.e., the dependencies between them. For example, the central phase of development is naturally suited to rapid rates of progress since that is when the largest number of problems are visible. Letting V(t) be the expected size of this set of visible (available for solving) problems at time t, Parr's model yields the equation

$$V(t) = \frac{Ae^{-\gamma\alpha t}}{(1 + Ae^{-\gamma\alpha t})^{(\gamma + 1/\gamma)}}$$

where

α is the proportionality constant relating the rate of progress and the expected size of the visible set

A is a measure of the amount of work done on the project before the project officially starts

γ is a structuring index which measures how much the development process is formalized and uses modern techniques.

The curve represented by V(t) differs from the Rayleigh/Norden curve for $y'(t)$ in two important ways. The Rayleigh curve is constrained to go through the origin; the Parr curve is not. Making $y'(0) = 0$ corresponds to setting an official start date for the project. Before that point, the effort expended on the project is assumed to be minimal. In reality, there is often a good deal of work done before that date, including such activities as requirements analysis and feasibility studies. In Putnam's environment, these were handled by a separate organization and could be ignored. Another factor that affects the problem space is past experience in the application area, or even more tangible is the influence of design or code taken from past projects. All of these have the effect of structuring the problem space at the beginning, so that more progress can be made early. The Parr curve accounts for this; the Putnam curve does not. See Fig.1 for a comparison of the two curves.

A second distinction between the two curves is the flexibility of where the point of maximum effort can come. By using a structuring index greater than one, this point of maximum effort can be delayed almost to acceptance testing and effort could still be drastically reduced before project completion. With the Rayleigh curve, a late point of maximum effort constrains the curve to have a slow buildup and almost no decay at the end.

Parr does not say how to estimate the parameters for V(t) in terms of data the project manager would have on hand. This is a problem in doing resource estimation currently, but the model could use the existing resource allocation schedule, based on early data points, to predict the latter part of the curve. The Parr model is only currently being tested on real

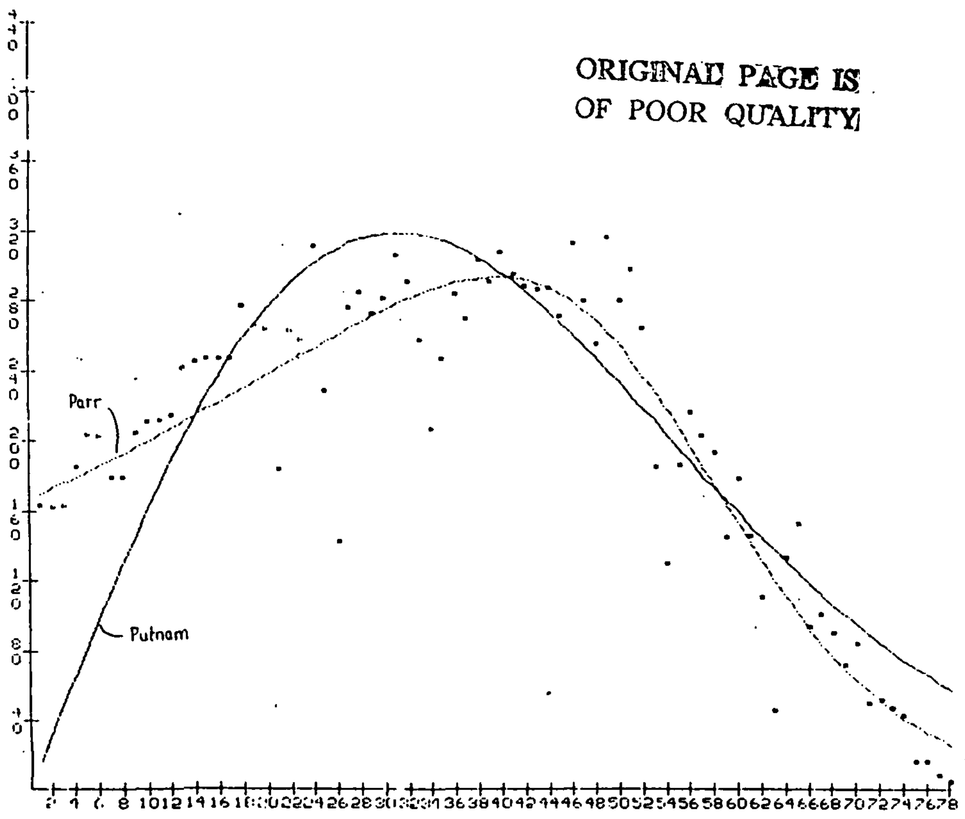


Fig.1 Weekly direct labor resources (manhours)
PARR vs PUTNAM

software for the first time and the results are not yet available. The Rayleigh model, on the other hand, has been used in many environments and has been quite successful on the whole.

Single variable, theoretical. The two previous theoretical models may be thought of as macro models in that the estimate of staffing levels relies on process oriented issues, such as total effort, schedule constraints, and the degree that structured methodology is used. Product oriented issues, such as source code, are not a factor. Most of the other models are less macro oriented in that they consider product characteristics, such as lines of code and input/output formats. In this section, we will discuss another type of theoretical model, based upon lower level aspects of the product, which we will call a micro model. The particular model discussed here deals with the idea that some basic relationships hold with regard to the number of unique operators and operands used in solving a problem and the eventual effort and time required for development. This notion was proposed by Halstead as part of his software science (12). Here there is only one basic parameter--size--measured in terms of operators and operands. The model transcends methodology and environmental factors. Most of the work in this area has dealt with programs or algorithms of module size rather than with entire systems, but that appears to be changing.

In the language of software science, measurable properties of algorithms are

n_1	number of unique or distinct operators in an implementation
n_2	number of unique or distinct operands in an implementation
$f_{1,j}$	number of occurrences of the j^{th} most frequent operator, $j = 1, 2, ..n_1$
$f_{2,j}$	number of occurrences of the j^{th} most frequent operand, $j = 1, 2, ..n_2$

then the vocabulary of an algorithm is

$$n = n_1 + n_2$$

and the implementation length is

$$N = N_1 + N_2$$

where

$$N_1 = \sum_{j=1}^{n_1} f_{1,j} \quad , \quad N_2 = \sum_{j=1}^{n_2} f_{2,j} \quad , \quad N = \sum_{i=1}^{2} \sum_{j=1}^{n_i} f_{i,j}$$

Based only on the unique operators and operands, the concept of program length N can be estimated as

$$\hat{N} = n_1 \log_2 n_1 + n_2 \log_2 n_2$$

$\hat{N}$ is actually the number of bits necessary to represent all things that exist in the program at least once, i.e., the number of bits necessary to represent a symbol table. Over a large set of programs in different environments, it has been shown that $\hat{N}$ approximates N very well.

To measure the size of an algorithm, software science transcends the variation in language and character set by defining algorithm size (volume) as the minimal number of bits necessary to represent the implementation of the algorithm. For any particular case, there is an absolute minimum length for representing the longest operator or operand name expressed in bits. It depends upon n, e.g., a vocabulary of 8 elements requires 8 different designators, or $\log_2 8$ is the minimal length in bits necessary to represent all individual elements in a program. Thus, a suitable metric for size of any implementation of any algorithm is $V = N \log_2 n$, called volume.

The most succinct form in which an algorithm can be expressed requires a language in which the required operation is already defined and implemented. The potential volume, V*, is defined as

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

but minimal form implies $N_1^* = n_1^*$ and $N_2^* = n_2^*$ because there should be no repetition. The number of operators should consist of one distinct operator for the function name and another to serve as an assignment or grouping symbol so $n_1^* = 2$. Thus, $V^* = (2 + n_2^*) \log_2 (2 + n_2^*)$ where n_2^* represents the number of different input/output parameters. Note: V* is considered a useful measure of an algorithm's content. It is roughly related to the basic GRC model concept of input/output formats. In fact, the GRC equation for man months of the project (MM_{PROG}) is an exponential relationship between MM_{PROG} and an estimate of V*.

The level of the implementation of a program is defined as its relation to its most abstract form, V*, i.e., $L = \frac{V^*}{V}$. $L \leq 1$ and the most succinct expression for an algorithm has a level of 1. $V^* = L \times V$ implies that when the volume goes up the level goes down. Since it is hard to calculate V*, an approximation for L, $\hat{L}$, is calculated directly from an implementation

$\hat{L} = \frac{2n_2}{n_1 N_2} \sim L$. The reciprocal of level is defined as the difficulty, $D = 1/L$, which can be viewed as the amount of redundancy within an implementation.

Based on these primitives, formulas for programming effort (E) and time (T) are derived. Assuming the implementation of an algorithm consists of N selections from a vocabulary of n elements and that the selection is non-random and of the order of a binary search (implying $\log_2 n$ comparisons for the selection of each element), the effort required to generate a program is $N \log_2 n$ mental comparisons (this is equal to the volume (V) of the program). Each mental comparison requires a number of elementary mental discriminations where this number is a measure of the difficulty (D) of the task. Thus, the total number of elementary mental discriminations E required to generate a given program should be $E = V * D = V/L = V^2/V^*$. This says the mental effort required to implement any algorithm with a given potential volume should vary with the square of its volume in any language. E has often been used to measure the effort required to comprehend an implementation rather than produce it, i.e., E may be a measure of program clarity.

To calculate the time of development, software science uses the concept of a moment, defined by the psychologist Stroud as the time required by the human brain to perform the most elementary discrimination. These moments have been shown to occur at a rate of 5 to 20 per second. Denoting moments (or Stroud's number) by S, we have $5 \leq S \leq 20$ per second. Assuming a programmer does not "time share" while solving a problem, and converting the effort equation (which has dimensions of both binary digits and discriminations) we get

$T = \frac{E}{S} = \frac{V}{SL} = \frac{V^2}{SV^*}$. Halstead empirically estimated S = 18 for his environment, but this may vary from environment to environment.

Software science metrics have been validated in a variety of environments but predominantly for module size developments.

<u>Other resources</u>. In what has been stated so far, resource expenditure and estimation have been predominantly computed in terms of effort. The formula for cost may be a simple multiplication of the staff months times the average cost of a staff member or it may be more complicated. It may include some difference for the cost of managers versus the cost of programmers versus the cost of support personnel whose role varies across the life cycle (13).

The schedule may be derived based upon historical data, with effort allocated to different activities based upon the known percentages or it may be dictated by the model itself, as with the Rayleigh curve. However, the dynamic models generate what they consider the ideal staffing conditions which may not be the actual ones available. Thus, in fitting actual effort to the estimated or proposed effort, some decisions and tradeoffs must be made.

Computer time is yet another resource. Unfortunately, none of the above models treats this within the same formula. In general, they have a separate formula for computer time again based upon computer use in similar projects. These models vary from a simple table type model (6) to some very sophisticated probability distribution based on reliability modeling for phases of the development, such as testing (14).

<u>Changes and Errors</u>

There are process aspects other than resource expenditures that provide information about managing and engineering the process and the product. One such aspect is the changes and errors generated during development or maintenance. Monitoring the changes in the software provides a measure of level of effort to get the product in order. If we can classify the types of changes that occur or their source of origin, we can categorize the environment and gain insight into how to manage or minimize the effect of particular types of changes. For example, suppose the user is generating a series of major changes at a continual rate. This may provide management with the information it needs to reclassify the environment from its original one to a more complex one, permitting modification of the cost parameters in the resource estimation model and a reestimation of cost part way through the project. It could also provide management with the necessary insight to change the development approach or methodology to one that is more insensitive to externally generated change, such as some incremental development approach.

Monitoring errors provides information with regard to the quality of the product. A product developed with only a few errors or with errors found early and an error rate decreasing during development and testing will warrant more confidence in its quality. Keeping track of the time to find and fix errors gives insights into cost. Knowing the types of errors being made helps in focusing attention to particular problems during the code-reading and design-review sessions.

Program evolution measures. Belady and Lehman (15) have examined the changes occurring in software during maintenance and derived a set of laws for program evolution. Based on such parameters as size of the system, number of modules added, deleted or changed, the release data, manpower, machine time and cost, they derived the following laws:

1. Law of continuing change. A system that is used undergoes continuing change until it is judged more cost effective to freeze and recreate it.

2. Law of increasing entropy. The entropy of a system (its unstructuredness) increases with time, unless specific work is executed to maintain or reduce it.

3. Law of statistically smooth growth. Growth trend measures of global system attributes may appear to be stochastic locally in time and space, but, statistically, they are cyclically self-regulating with well-defined long-range trends.

These laws can be demonstrated by using the following metrics:

RSN, the release number

D_r, the age of system at release R

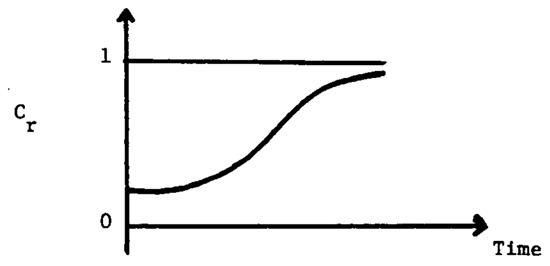
I_r, the time between releases R-1 and R

M_r, the number of modules in the system

MH_r, the number of modules handled during release interval I_r (estimator of activity undertaken in each release)

$HR_r = MH_r/I_r$, the handle rate

$C_r = MH_r/M_r$, the complexity which is the fraction of released system modules that were handled during the course of the release R.



C_r has been observed to be monotonically increasing and approaching unity over time (for OS 360, approximately 20 releases over 10 years).

Using these metrics, management can predict when it is too costly to modify a system, i.e., when it is cheaper to redesign than make the next change. It can also determine whether enough effort is being devoted to keep future changes at a reasonable cost.

Program-changes. Dunsmore and Gannon have proposed a measure called program-changes which correlates very highly with errors (16). A program-change is a textual revision in the source code of a module during the development period. One program-change should represent one conceptual change to the program. Thus, a program-change is defined as one or more changes to a single statement, one or more statements inserted between existing statements, or a change to a single statement followed by the insertion of new statements. On the other hand, the following are not counted as program-changes: the deletion of one or more existing statements, insertion of standard output statements or special compiler-provided debugging directives, and insertion of blank lines or comments. Basili and Reiter showed that program-changes were minimal when a good software development method was used (17).

Error-day. An error-based measure of product quality was proposed by Mills (18) which he called the error-day. The motivation is that the longer an error remains in the system the more expensive and less reliable it is. The error-day measure is simply the sum over each error of the number of days it has existed within a system. It weights errors by their duration in the system. Clearly, a low error-day count is an indicator of a well-engineered program. This measure could be automated by using the concept of program-changes and plotting them against time.

Job-steps. An indication of the amount of effort expended in development can be the number of computer accesses or job-steps. A computer job-step is a single programmer-oriented activity performed on a computer at the operating system command level, which is basic to the development effort and involves nontrivial expenditures of computer or human resources. Typical job-steps might be text editing, module compilation, link editing, and program execution. Basili and Reiter (17) found job-steps to be a serious differentiator of development environments, and that good methodology leads to a smaller number of job-steps.

There exist many other measures of the software development process. The interested reader is referred to some general references in the literature, e.g., Curtis (19), Mohanty (20), Belady (21).

PRODUCT MEASURES

Actually, all the previous measures could have been considered measures of the product. If a product takes a long time or a large effort to develop, we may consider it a complex product. If there were lots of errors found at the tail end of product development or if the rate of finding errors was increasing every day, we would say the quality of the product was very low. However, each of those indicated as much, if not more, about the process than the product.

The measures discussed in this section are probes into the product. They are taken at a discrete point in time, usually on the final deliverable product. Even though examining the changes in value of the metrics on the product over time could be very informative with regard to the process, we will classify them as product measures. We categorize these measures with respect to size, structure and reliability.

Size

The size of a product is a simplistic measure and easy to calculate. It is a reasonable indicator of the amount of work expended and correlates well with effort. Size metrics are used for cost estimation, comparison of products, and for measures of productivity. Although it may be a basic ingredient in effort and productivity measures, it must be modified by many other factors, such as reliability and complexity. These measures will be treated in subsequent sections.

The most common measure of size is lines of code.

However, what gets measured depends to a great extent on our interests. For example, if we are interested in measuring effort, then source lines including comments and data are a reasonable measure and have been used in several studies (1, 2). If we are interested in function size, a better approximation may be executable statements. If our interest is in comparing the size of resulting products for operational use, a common denominator is number of machine language instructions. Clearly, there is little agreement on the appropriate measure of lines of code and the choice should depend upon the issue under consideration. It is important in reading the literature that we clearly understand which measure of size is being used, since the authors do not always make it clear.

Another measure of size is to treat units larger than lines of code. One common unit is the module. Modules are used in the measures of Belady and Lehman (21) and were shown to be reasonable measures for cost estimation by Freburger and Basili (2). Smaller units, such as procedures or functions, were used by Basili and Reiter (17). Again, the choice is dependent upon the purpose of the measure. For estimation, it is sometimes easier to predict the number of modules rather than the number of lines. However, comparison may be difficult since there is no standard definition of module.

On the other end of the size spectrum is the number of operators and operands as defined in software science by Halstead (12). More specifically, the length and volume measures are potential measures for size of an implementation and size of the function, respectively. There have been several studies that support these metrics as reasonable approximations to what they purport to measure. They make good metrics for comparison and possible evaluation, but there is potential for using them for estimation also.

Structure

The structure of a program is often a good indicator of whether that product is well designed, understandable, and easy to modify. Structure measures are often proposed as measures of the complexity of the product. In examining structure, we may be concerned with the control structure, the data structure, or a mixture of the two.

Control structure measures. The simplest control structure metric is the number of decisions (17) as measured by the number of constructs that represent branches in the flow of control, such as if then else or while do statements. There is a basic belief that the more control flow branching there is in a system the more complex it is. A variation of this measure is the relative percentage of control flow branching, i.e., the number of decisions divided by the number of executable statements. Early studies by Aron (7) showed that varying levels of this type of complexity could account for a nine to one difference in productivity.

A more refined measure of control complexity is cyclomatic complexity as proposed by McCabe (22). The cyclomatic complexity of a graph is defined as the number of edges minus the number of nodes plus the number of connected components, and is equal to the minimum number of basic paths from which all other paths may be constructed. Given a program in which all statements are on a path from the entry node to an exit node, the cyclomatic complexity can be defined as the number of predicates plus the number of segments. A predicate is defined as a simple Boolean expression governing the flow of control and a segment is defined as an individual routine (procedure or function).

The measure originated as a count of the minimum number of program paths to be tested. This is one quantitative measure of a program's complexity. The measure is usually applied at the module level and McCabe proposed a cyclomatic complexity of ten as an upper bound for the safe range with regard to the complexity of a module. Several variations of the basic cyclomatic complexity measure have been studied by Basili and Reiter (23). They evaluated their sensitivity to different software development environments with reasonable success. They have also defined some approaches to using the measure at the product level rather than the module level in a way that is reasonably insensitive to system modularization.

Other measures of control complexity involve the weighting of various types of control structures as to whether they are simple or complex, where simple means easy to read and prove correct based upon the graph structure. For example, single-entry single-exit program graphs that contain a single predicate node are easier to understand and abstract from than more complicated graph structures. Thus, one approach would be to weight various graph structures based upon this complexity. This type of measure requires a more detailed analysis of the program structure than does the cyclomatic complexity measure, but tends to be a deeper measure of control flow and can include other complexity factors, such as nesting level. One such measure is essential complexity (22), which assigns every program using only structured programming control structures a complexity of one.

Data structure measures. Data structure metrics try to measure the complexity of the program structure by the way the data is used, organized, and allocated. Clearly, the simpler the reader's ability to abstract the use of data the easier the program will be to understand and modify. Several measures have been used for evaluating the structuring of the data in a program and a few will be discussed here.

The segment-global usage pair metric (24) attempts to measure the goodness of the use of globals in the program. A segment-global usage pair (p, r) is an instance of a global variable r being used by a segment p (i.e., r is either modified or accessed by p. Each usage pair represents a unique "use connection" between a global and a segment. Let actual usage pair (AUP) represent the count of realized usage pairs, i.e., r is actually used by p. Let possible usage pair (PUP) represent the count of potential usage pairs, i.e., given the program's globals and their scopes, the scope of r contains p so that p could potentially modify or access r. This represents a worst case. Then the relative percentage usage pairs (RUP) is RUP = AUP/PUP and is a way of normalizing the number of usage pairs relative to the problem structure. The RUP metric is an empirical estimate of the likelihood that an arbitrary segment uses an arbitrary global.

The data binding metric (24, 25) is an attempt at measuring the inter-relationship of modules or segments within a program. A segment-global-segment data binding (p, r, q) is an occurrence of the following: (1) segment p modifies global variable r, (2) variable r is accessed by segment q, and (3) $p \neq q$. The existence of a data binding (p, r, q) implies that q is dependent on the performance of p because of r. Binding (p, r, q) does not equal binding (q, r, p). (p, r, q)

ORIGINAL PAGE IS
OF POOR QUALITY

represents a unique communication path between p and q and the total number of data bindings represents the degree of a certain kind of "connectivity," i.e., between segment pairs via globals, within a complete program. Let actual data bindings (ADB) represent the absolute number of realized data bindings in the program, i.e., the realized connectivity, and possible data bindings (PDB) represent the absolute number of potential data bindings given the program's global variables and their declared scope (i.e., same worst case). Then we can normalize the number of data bindings by calculating the relative percentage RDB = ADB/PDB. This gives some relative measure of the amount of information exchanged in the program.

A measure of the amount of data required to be understood by the programmer while reading a program is span (26). A span is the number of statements between two consecutive textual references to the same identifier. Thus, for n appearances of an identifier in the source text, n-1 spans are measured. All appearances are counted except those in declare statements. If the span of a variable is greater than one hundred statements, then one new item of information must be remembered for a hundred statements until it is read again. The complexity of the program would be the number of spans at any point, i.e., the amount of data the reader must be aware of when reading any particular statement.

Control and data structure measures. There are models of structure that address the integration of control and data flow. One such model is slicing (27). Informally, slicing reduces a program to a minimal form which still produces a given behavior for a subset of the data. The desired behavior is specified as a projection from the program's original behavior. For instance, if a program computes values for variables X, Y, and Z, then one projection might be the value of X at program termination. The minimal program is obtained by eliminating program statements which do not affect the projected behavior. The result is a smaller program which contains only those statements from the original program which affect the selected behavior.

There are several possible metrics based on program slicing. These include (1) coverage, the ratio of slice length to program length; (2) overlap, a measure of the sharing of statements among different slices; (3) clustering, the percentage of statements in the slice which were adjacent in the original program; (4) parallelism, the number of almost disjoint slices; and (5) tightness, the ratio of statements found in every slice to total statements in the original program. Each of these metrics gives some view of the complexity of the program with respect to the control and data flow.

Reliability

Measuring the reliability of a product may involve an analysis of the (1) distribution or classification of errors, or (2) execution of the product in a testing or operational environment. Metrics involving the distribution of errors can include the program changes and error-day metrics discussed earlier. Other metrics involve distributions, such as fixes per line of code, fixes per phase, errors per person hour, errors per type of change causing the error, fixes per detection and correction technique, etc. Weiss (28) has studied various distributions in evaluating a development methodology by showing a profile of the error distributions made when using the methodology. Endres (29) used error classification schemes to analyze the reliability of a release of an operating system.

With regard to the operation of the program, several reliability models have been proposed in the literature (14, 30, 31, 32). Software reliability here is defined as the probability that a given software program operates for some time period without software error which is detectable by executing the code on the machine for which it was designed, given that it is used within design limits. Reliability measurement can be done for evaluation purposes as well estimation purposes. The models measure reliability as a function of calendar time, computer usage or accumulated man hours and require parameters, such as the error detection rate and the total number of errors in the system, before testing. These estimates can be based on theoretical assumptions or historical data.

A particular reliability model due to Shooman (30) is based upon a set of assumptions, such as (1) the operational software errors occur due to occasional traversing of a portion of the program in which a hidden software bug is lurking; (2) the probability that a bug is encountered in the time interval Δt, after t successful hours of operation is proportional to the probability that any randomly chosen instruction contains a bug, i.e., the fractional number of remaining bugs ϵ_r. Then the probability of a failure during time interval $(t, t + \Delta t)$, given no failures have occurred up until t is proportional to the failure rate z(t) (hazard function). Thus, the probability of failure in interval Δt, given no previous failure, is $P\ (t < t_f \leq t + \Delta t \mid t_f > t) = z(t) * \Delta t = K\epsilon_r\ (\tau)\ \Delta t$ where t_f is operating time to failure, K is an arbitrary constant, τ is the debugging time in man months, t is the operating time in hours. K can be estimated by examining the history of errors detected, e.g.,

$$K \sim \frac{\#\text{ catastrophic errors detected.}}{\text{total }\#\text{ errors detected}}$$

The probability of no system failure in the interval (0, t) is given by the reliability function

$$R(t) = e^{-\int_0^t z(x)\,dx}$$

assuming reliability is related to the failure rate. Assuming K and $\epsilon_r\ (\tau)$ are independent of operating time t we get

$$R(t) = e^{-\{K\epsilon_r(\tau)\}t} = e^{-\delta t}$$

$$= e^{-K\{E_T/I_T - \epsilon_c(\tau)\}t}$$

where ϵ_c is the number of corrected errors, E_T is the total number of initial bugs in the program and I_T is the number of instruction in the program. This implies the probability of successful operation without software bugs is an exponential function of operating time.

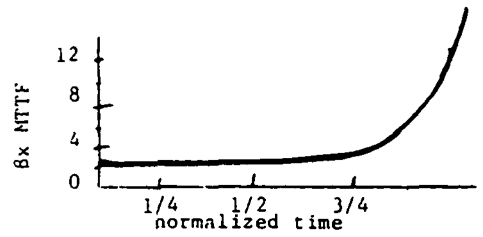
A simplier way to summarize the results of the reliability model is to compute the mean time to (software) failure, MTTF using the reliability function

$$\text{MTTF} = \int_0^\infty R(t)dt = \frac{1}{K\epsilon_r(\tau)} = \frac{1}{K\{E_T/I_T - \epsilon_c(\tau)\}}$$

If the error correction rate ρ_o is constant, then $\epsilon_c\ (\tau) = \rho_o\ (\tau)$ and

$$\text{MTTF} = \frac{1}{K\left[\frac{E_T}{I_T} - \rho_o\tau\right]} = \frac{1}{\beta\ (1 - \alpha\ \tau)}$$

ıhere $\beta = \frac{E_T}{I_T} K$ and $\alpha = \frac{\rho_0 \; I_T}{E_T}$



Note the most improvement in MTTF occurs during the last quarter of debugging.

Other models are based upon different assumptions, but all yield some measure of the reliability of the product.

The reader interested in other product measures s again referred to some general references in the literature (19, 20, 21).

RODUCT MEASURES ACROSS TIME

As mentioned earlier, measures can be taken once on the final product or at discrete intervals through-ut the life cycle. In this latter approach, metrics an be used to monitor the stability and quality of the product. By re-evaluating the metrics periodically, we can see if the product is changing its charac-er in any way. It can provide feedback during develop-ent and maintenance. For example, if we find that ver a period of time more and more control decisions have entered the system, then something may have to be done to counteract this change in character.

This approach is a way of providing a relativistic evaluation of the product. As such, it is easier to understand than an absolute measure. That s, it may be more informative to know that each hange we make in the system increases the complexity of the system, than to know the total complexity of the system is some specific number. Here we need nly compare the values of the metrics with values of he metrics on earlier versions of the system. The rawback to an absolute measure is that we have nothing to compare it to.

ATA COLLECTION

One major concern with performing measurement is the ability to collect reliable data. Before we begin ollecting data, however, we must first understand he various factors that characterize our environment. e must isolate those factors we hope to control, measure, and understand so that we may analyze their effect.

With regard to the actual data collection process, there are various approaches. Data collection can be automated, meaning there is no interference to the evelopers, or non-automated, meaning the data is ollected from the developers using forms or interviews. Automated data collection tends to be more reliable and can be done without the participants being aware f what specific activities and factors are being udied. Reporting forms and interviews can provide ore detailed insights into the process and give a level of information that is not available in an automated collection process, e.g., insights into the kinds of errors committed.

Clearly, the data collected should be driven by the models and metrics we are interested in using; however, it doesn't hurt to add other data which may give us information about refining and modifying those models and metrics. All the data collected should be entered into a data base and validated, as much as possible, for easy reference and access.

A first step in the validation of forms is a review of the forms as they are handed in; someone connected with the data collection process should ensure that the appropriate forms have been handed in and that the appropriate fields have been filled out. The data should be entered into the data base through a program that checks the validity of the data format and rejects data out of the appropriate ranges. For example, this program can assure that all dates are legal dates and that system component names and programmer names are valid for the project by using a prestored list of component and programmer names.

Ideally, all data in the data base should be reviewed by individuals who know what the data should look like. Clearly, this is expensive and not always possible. However, several projects should be reviewed in detail and the number and types of discrepancies kept so that bounds can be calculated for the unchecked data. This allows data to be interpreted with the appropriate care.

Another type of validity check is to examine the consistency of the data base by comparing redundant data. For example, if effort data is collected both at the budget level and at the individual programmer level, there should be a reasonable correlation between the two total efforts. Another approach is to use cluster analysis to look for patterns of behavior that are indicative of errors in filling out the forms. For example, if all the change report forms filled out by a particular programmer fall into one cluster, it may imply that there is a bias in the data based upon the particular programmer.

Data collection is a serious problem, especially on large programming projects involving characteristically different environments. One set of forms may not be enough to capture what is happening across all environments. However, if we are to use this data in models and metrics, we need to know how valid that data is in each case so as to avoid improper conclusions.

CONCLUSION

Having fit the models to the data, we must analyze and interpret their results carefully. As stated earlier, we must understand the environmental parameters under which the project was developed. We must know the assumptions, strengths and weaknesses of the models in order to interpret the results for the particular project. Our level of confidence in the particular model or metric should be based upon the level to which the model or metric has been tested. If the results support our intuition, we understand what the model means in our environment; if not, understanding the model's shortcomings can yield insights into the model and our environment.

Quantitative support can be an excellent aid and risk reducer in making a difficult management or engineering decision. An organization should build up its knowledge and expertise in quantitative analysis of software development. In this way, confidence in the various models and metrics can be acquired through direct experience.

Acknowledgement: The author would like to thank John Bailey, John Beane, David Hutchens, and Robert Reiter for their insightful review of this article. *

REFERENCES

(1) Walston, C. and Felix, C., "A Method of Programming Measurement and Estimation," IBM Systems Journal 16, Number 1, 1977.

(2) Freburger, Karl and Basili, Victor, "The Software Engineering Laboratory: Relationship Equations," University of Maryland Technical Report TR-764, May 1979.

(3) Lawrence, M. J. and Jeffery, D. R., "Inter-organizational Comparison of Programming Productivity," Department of Information Systems, University of New South Wales, March 1979.

(4) Boehm, Barry W., Draft of book on Software Engineering Economics, to be published.

(5) Doty Associates, Inc., Software Cost Estimates Study, Vol. 1, RADC TR 77-220, June 1977.

(6) Wolverton, R., "The Cost of Developing Large Scale Software," IEEE Transactions on Computers 23, No. 6, 1974.

(7) Aron, J., "Estimating Resources for Large Programming Systems," NATO Conference on Software Engineering Techniques, Mason Charter, N. Y. 1969.

(8) Carriere, W. M. and Thibodeau, R., "Development of A Logistics Software Cost Estimating Technique for Foreign Military Sales," General Research Corporation, Santa Barbara, California, June 1979.

(9) Putnam, L., "A General Empirical Solution to the Macro Software Sizing and Estimating Problem," IEEE Transactions on Software Engineering 4, No. 4, 1978.

(10) Norden, Peter V., "Useful Tools for Project Management," Management of Production, M. K. Starr (Ed.) Penguin Books, Inc., Baltimore, Md. 1970, pp. 77-101.

(11) Parr, Francis N., "An Alternative to the Rayleigh Curve Model for Software Development Effort," IEEE Transactions on Software Engineering, May 1980.

(12) Halstead, M., Elements of Software Science, Elsevier North-Holland, New York, 1977.

(13) Basili, Victor R. and Zelkowitz, Marvin V., "Analyzing Medium Scale Software Developments," Third International Conference on Software Engineering, Atlanta, Georgia, May 1978.

(14) Musa, John D., "A Theory of Software Reliability and Its Application," IEEE Transactions on Software Engineering, Vol. SE1, No. 3, pp. 312-327.

(15) Belady, L. A. & Lehman, M. M., "A Model of Large Program Development," IBM Systems Journal, 1976, 15(3).

(16) Dunsmore, H. E. & Gannon, J. D., "Experimental Investigation of Programming Complexity," Proc. ACM-NBS Sixteenth Annual Technical Symposium: Systems and Software, Washington, D. C., June 1977, pp. 117-125.

(17) Basili, V. R. and Reiter, R. W. Jr., "An Investigation of Human Factors in Software Development," Computer Magazine, December 1979, pp. 21-38.

(18) Mills, H. D., "Software Development," IEEE Transactions, Syst. Eng. 2, 4 (1976), pp. 265-273.

(19) Curtis, Bill, "In Search of Software Complexity," Proceedings of the Workshop on Quantitative Models of Software Reliability, Complexity, and Cost, New York, IEEE 1979.

(20) Mohanty, S. N., "Models and Measurements for Quality Assessment of Software," ACM Computing Surveys, 1979, 11, pp. 251-275.

(21) Belady, L. A., "Complexity of Programming: A Brief Summary," Proceedings of the Workshop on Quantitative Models of Software Reliability, Complexity, and Cost, New York: IEEE, 1979.

(22) McCabe, T. J., "A Complexity Measure," IEEE Transactions on Software Engineering, 1976, 2, 308-320.

(23) Basili, V. R. & Reiter, R. W., Jr., "Selecting Automated Measures of Software Development," Proceedings of the Workshop on Quantitative Models of Software Reliability, Complexity, and Cost, New York, IEEE 1979.

(24) Basili, V. R. & Turner, A. J., SIMPL-T, A Structured Programming Language, Paladin House Publishers, Geneva, Ill. 1976.

(25) Stevens, W. P., Myers, G. J., and Constantine, L. L., "Structured Design," IBM Systems Journal, Vol. 13, No. 2, 1974 pp. 115-139.

(26) Elshoff, "An Analysis of Some Commercial PL/1 Programs," IEEE-TSE June 1976.

(28) Weiss, David, "Evaluating Software Development by Error Analysis: The Data from the Architecture Research Facility," Journal of Systems and Software, Vol. 1, No. 1, 1979.

(27) Weiser, Mark, "Program Slices: Theoretical, Psychological, and Practical Investigations of An Automatic Program Abstraction Method," Ph.D. Thesis, University of Michigan 1979.

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

(30) Shooman, M. L. "Software Engineering: Reliability Design and Management," Note of Course EE909, Polytechnic Inst. of New York, Brooklyn, N.Y., 1976.

(31) Littlewood, B., "How to Measure Software Reliability, and How Not to . . .," in Proc. 3rd Int. Conf. Software Engineering, May 1978, pp. 37-45.

(32) Goel, A. L. & Okumoto, "A Markovian Model for Reliability and Other Performance Measures of Software Systems," Proceedings of the National Computer Conference, pp. 769-774 (1979).

*Research was supported in part by NASA Grant NSG5123