Discussed are observations made on the development of OS/360 *and its subsequent enhancements and releases. Some modeling approaches to organizing these observations are also presented.*

A model of large program development

by L. A. Belady and M. M. Lehman

As a need for a discipline of software engineering has been recognized, the design, implementation, and maintenance of computer software has come into the forefront. The formulation of concepts of programming methodology, exemplified by Dijkstra's structured programming,[1] strikes at the roots of the problem. The realization is that a program, much as a mathematical theorem, should and can be provable. Recognition that a program can be proved correct as it is developed and maintained,[2] and before its results are used, may ultimately change the nature of the programming task and the face of the programming world. Clearly these developments are of fundamental importance. They appear to point to long-term solutions to problems that will be encountered in creating the great amount of program text that the world appears to require. But even though progress in mastering the science of program creation, maintenance, and expansion has also been made, there is still a long way to go.

the system approach

Such progress as is currently being made stems primarily from the personal involvement of researchers and developers in the programming process at a detailed level. Often they tackle a single problem area: algorithm development, language, structure, correctness proving, code generation, documentation, or testing. Others view the process as a whole, yet they are primarily concerned with the individual steps that, together, take one from concept to computation. Still this type of study is essential if real insight is to be gained and progress made.

Figure 1 Growth trends of system attribute counts with time

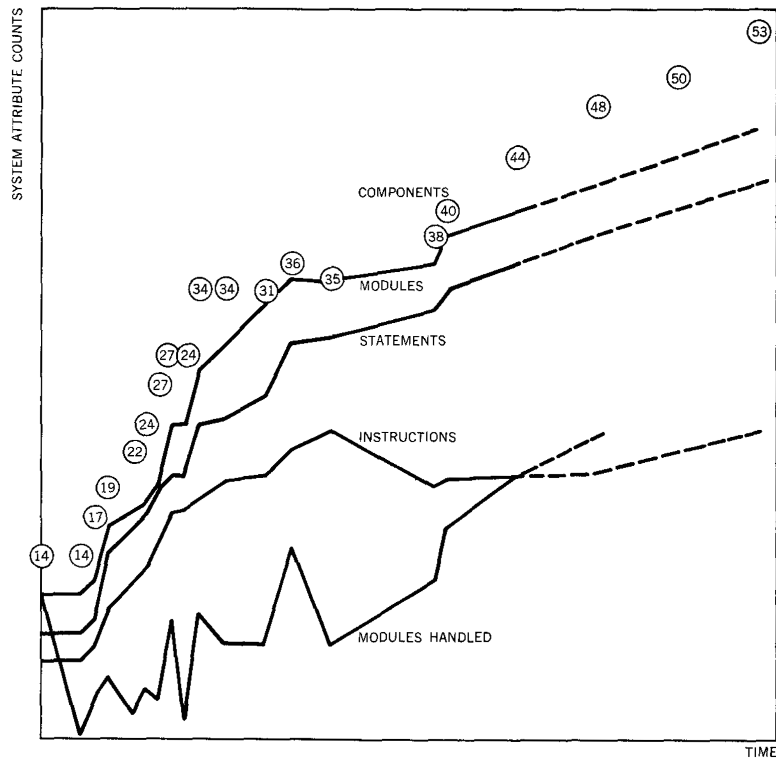
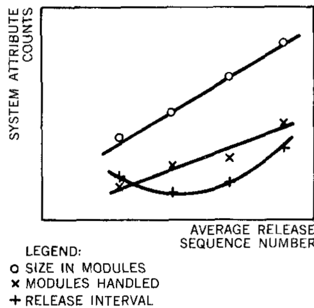


Figure 2 Average growth trends of system attributes



The scientific method has made progress in revealing the nature of the physical world by pursuing courses other than studying individual phenomena in exquisite detail. Similarly, a system, a process, or a phenomenon may be viewed from the outside, by acts of observing; clarifying; and by measuring and modeling identifiable attributes, patterns, and trends. From such activities one obtains increasing knowledge and understanding, based on the behavior of both the system and its subsystems, the process and its subprocesses.

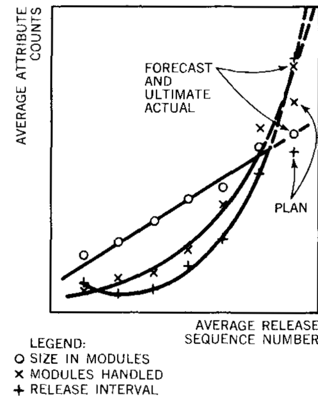
Starting with the initial release of OS/360 as a base, we have studied the interaction between management and the evolution of OS/360 by using certain independent variables of the improvement and enhancement (i. e., maintenance) process. We cannot say at this time that we have used all the key independent variables. There is undoubtedly much more to be learned about the variables and the data that characterize the programming process. Our method of study has been that of regression—outside in—which we have termed "structured analysis." Starting with the available data, we have attempted to deduce the nature of consecutive releases of OS/360. We give examples of the data

that support this systematic study of the programming process. Again, however, we wish to emphasize that this study is but the beginning of a new approach to analyzing man-made systems.

the programming process

The authors have studied the programming process[3] as it pertains to the development of OS/360, and now give a preliminary analysis of some project statistics of this programming system, which had already survived a number of versions or releases when the study began. The data for each release included measures of the size of the system, the number of modules added, deleted or changed, the release date, information on manpower and machine time used and costs involved in each release. In general there were large, apparently stochastic, variations in the individual data items from release to release.

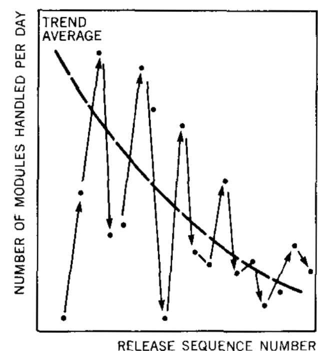
Figure 3 Average growth trends of system attributes compared with planned growth



All in all, the data indicated a general upward trend in the size, complexity, and cost of the system and the maintenance process, as indicated by components, modules, statements, instructions, and modules handled in Figure 1. The various parameters were averaged to expose trends. When the averaged data were plotted as shown in Figure 2, the previously erratic data had become strikingly smooth.

Some time later, additional data were plotted as shown in Figure 3 and confirmed suspicions of nonlinear—possibly exponential—growth and complexity. Extrapolation suggested further growth trends that were significantly at odds with the then current project plans. The data were also highly erratic with major, but apparently serially correlated, fluctuations shown in Figure 4 by the broken lines from release to release. Nevertheless, almost any form of averaging led to the display of very clear trends as shown by the dashed line in Figure 4. Thus it was natural to apply uni- and multivariate regression and autocorrelation techniques to fit appropriate regression and time-series models to represent the process for purposes of planning, forecasting, and improving it in part or as a whole. As the study progressed, evidence accumulated that one might consider a software maintenance and enhancement project as a self-regulating organism, subject to apparently random shocks, but—overall—obeying its own specific conservation laws and internal dynamics.

Figure 4 Serial and average growth trends of a particular attribute



Thus these first observations encouraged the search for models that represented laws that governed the dynamic behavior of the metasystem of organization, people, and program material involved in the creation and maintenance process, in the evolution of programming systems.

laws of program evolution

It is perhaps necessary to explain here why we allege continuous creation, maintenance, and enhancement of programming systems. It is the actual experience of all who have been in-

volved in the utilization of computing equipment and the running of large multiple-function programs, that such systems demand continuous repair and improvement. Thus we may postulate the First Law of Program Evolution Dynamics.[4]

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

Software does not face the physical decay problems that hardware faces. But the power and logical flexibility of computing systems, the extending technology of computer applications, the ever-evolving hardware, and the pressures for the exploitation of new business opportunities all make demands. Manufacturers, therefore, encourage the continuous adaptation of programs to keep in step with increasing skill, insight, ambition, and opportunity. In addition to such external pressures for change, there is the constant need to repair system faults, whether they are errors that stem from faulty implementation or defects that relate to weaknesses in design or behavior. Thus a programming system undergoes continuous maintenance and development, driven by mutually stimulating changes in system capability and environmental usage. In fact, the evolution pattern of a large program is similar to that of any other complex system in that it stems from the closed-loop cyclic adaptation of environment to system changes and vice versa.

As a system is changed, its structure inevitably degenerates. The resulting system complexity and reduction of manageability are expressed by the Second Law of Program Evolution Dynamics.

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

This law too expresses vast experience, in part by data to be presented later in this paper. This, in turn, leads to the formulation of the Third Law of Program Evolution Dynamics.

III. *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.

The system and the metasystem—the project organization that is developing it—constitute an organism that is constrained by conservation laws. These laws may be locally violated, but they direct, constrain, control, and thereby regulate and smooth, the long-term growth and development patterns and rates. Observa-

tion, measurement, and interpretation of the latter can thus be used to plan, control, and forecast better the product of an existing process and to improve the process so as to obtain desired or desirable characteristics.

The "laws" that we are expounding upon have gradually evolved as we have pursued our study of the programming task. When we began our studies, observations led to the concept that we termed "programming systems growth dynamics."[5] We have now renamed this subdiscipline "programming evolution dynamics."

The remainder of this paper describes some of the statistical and formal models of the programming process that we have been able to develop by pursuing the consequences of the laws of programming evolution dynamics. It is our conviction that the extension of these studies can lead to an increasing understanding of the nature and dynamics of the programming process. Hence, studies such as these may yield significant advances in the ability to engineer software, i.e., to plan and control program creation and maintenance.

The process observed—a statistical model

The basic assumptions of programming evolution dynamics spring from viewing the program being implemented, enhanced, and maintained and its metasystem—the organization that generated and undertook the development of OS/360—as interacting systems. The evolutionary process and life cycle of a program are at least partially governed by the structural and functional attributes of both the program and the human organization. Their size, complexity, and numerous internal interactions suggest the use of statistical techniques for interpreting observed behavior.

Detailed studies of available data in conjunction with the almost universal experience of the programming community indicate that a large programming project has many of the properties of a multiple loop, self-stabilizing feedback system. The overall trend has been summarized in the previously discussed three laws that underlie the dynamics of evolution of large programs. The present section presents some of the accumulated numerical evidence derived from experience with OS/360—one model of one system from one environment.

available data

The project data presented here originate from OS/360, which is now some twelve years old. This system has been made available to an increasing number of users in a series of over twenty user-oriented releases. These releases have extended the capa-

bility of the operating system by correcting faults, improving performance, supporting new hardware, and by adding newly conceived functions.

These and other intermediate releases were assigned names or numbers as identifiers. Each release may, however, also be identified as a program that—with its documentation—forms an identifiable and stable text in an otherwise continuously changing environment. Assigning *Release Sequence Numbers* (*RSN*s) to versions receiving the same degree of exposure, yields a sequence of integers that forms a pseudo time measure in the sense of Cox and Lewis[6] that may be used to describe the time-dependent behavior of program evolution.

Of the releases considered, the first represents the culmination of the basic design and build (i.e. system integration) process. The iterative process that yields the specification, architecture, design, and the first implementation of a large program system differs significantly from subsequent maintenance and enhancement activity. In particular, there is at this stage no feedback of fault reporting or performance assessment by independent users. Hence data relating to that first release are not included in this analysis. The build process itself may, however, be studied by using data obtained periodically during the development activity.

Data from a second release were also unused because they were shown to represent a component development somewhat off the main stream. In the final analysis, the model and the plots to be presented are based on twenty-one sets of observations. This relatively small number of data points implies that extreme care must be exercised in interpreting the results of the statistical analysis. Subsequent data from the OS/360 augmented by data from other environments have generally confirmed our observations and conclusions.

observables of system evolution

The release sequence number (RSN) is taken as the first of the system evolution parameters. The second is the age of the system D_R at release with $RSN = R$. Equivalently, D_R is the inter-release interval I_R; in other words, the interval in days between releases with $RSN = R-1$ and R, respectively. A third available parameter M_R measures the size of the system in modules. We present the results of our analysis in terms of modules, though other size measures—such as numbers of components or instructions in the system—could also have been used. The suitability of the module stems from the fact that in OS/360 the concept of module—though imprecisely defined—represents at one and the same time a functional and implementation entity and, for execution, a unit of system generation and storage allocation.

A fourth parameter MH_R records the number of system modules that have received attention, i.e., those that have been handled during the release interval and, more specifically, during the integration process. We have used this as an initial estimator of the amount of activity undertaken in each release. The measure is imprecise, but represents the best available information over the entire sequence. From MH_R and I_R, in turn, we determine an estimate of the handle rate HR_R for the activity that produced the release with $RSN = R$.

From the very first beginnings of this study of the programming process,[5] it has been clear that the changing complexity of a system, as it is modified, plays a vital role in the aging process. Unfortunately there is no clear or unique understanding of what complexity is and how it can be defined and measured. The choice of complexity definition cannot, in fact, be disassociated from the use to which it is to be put. But complexity of the system, of the organization, and of each particular series of changes is fundamental to the maintenance and to the resultant aging process. Hence some measures of complexity must be established.

For the purposes of the present analysis, *complexity* C_R has been defined as the fraction of the released system modules that were handled during the course of the release with $RSN = R$. This definition is clearly inadequate. It does not separately measure the various independent complexity factors involved. It does not discriminate between system organization and the nature of the work undertaken. Nor does it measure the amount of activity involved. But at least it is a measure for which real data exist. Moreover the data give interpretable results. Hence $C_R = MH_R/M_R$ will suffice until better measures become available.

the present model

We have just identified five observable and measurable parameters of the programming process. Our hypothesis implies that these parameters do not vary independently, at least when viewed over a relatively long period of time. In fact, we have been able to determine, for example, four bivariate relationships among them. The complexity parameter, however, is derived from two of the others. Hence, on the basis of present data, we are entitled to fit only three independent functions. The fourth relationship, then, must be derived from the other three and tested for fit. As in all data fitting, the forms selected must also pass a test of conceptual reasonableness.

We stress that, in general, any statistical goodness of fit test is insufficient to establish any relationship as an element of the total model—as an expression of causal relationships—unless it can be convincingly interpreted in the light of one's insight into the process. Ultimately, it is only through the interplay and

iteration of observation, modeling, and interpretation that real progress can be made in understanding and mastering the large-scale programming process.

nature of the relationships

The statistically derived relationships to be presented here comprise a model of the programming process with respect to this system's life cycle. The relationships represent a simple, but recognizably incomplete, model of what is happening. In practice, the statistical model has been used to improve the planning for this particular system. With the insight gained from the model's development, further statistical and analytic models have been and will continue to be developed that may explain the process and eventually lead to the insight that permits improvement of process planning, control, and cost/effectiveness.

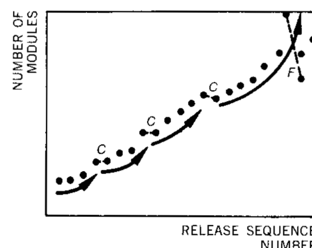
In the first instance, we must identify the global nature of the process as expressed in the relationships to be, or that have been, developed. The previously stated Third Law suggests that smooth long-term trends can be seen in the measures even if short-term behavior tends to be erratic. This is supported by the fact that we have been able to construct statistically significant relationships consisting of three parts: the first expresses the long-term, deterministic trend; the second describes short-term cyclic effects; and the third part expresses any system-relative stochastic influences on the process.

The stochastic influences arise, in part, from a certain arbitrariness in the selection of the new function and, therefore, new code to be included in any given release. It is influenced to a significant degree by user and management pressure, the availability of new hardware devices, and by business considerations that are not directly related to the internal dynamics of the process. Equally, the release target date, and hence the age of the system at the release point, is strongly influenced by factors external to the programming process.

The cyclic trends that we have observed in the data, and that have long been accepted on a heuristic basis by managers and observers of programming practice, may well contain the clue to current limitations of the process. In part, at least, inter-release effects arise from the interaction of repair and enhancement activity, particularly when they share common resources and are undertaken in parallel. It is probably the interplay between the levels and rates of the various activities and, in particular, their divisions at any given time between repair, functional improvements, and new capability additions that charts the fate of a programming system. Long-term trends, however, are perhaps of greatest significance in understanding the process and in foreseeing and influencing the future. It is this effect that we shall mainly stress in our analysis.

Figure 5 shows the size of OS/360 in modules plotted with respect to release sequence numbers. Relative to the nonuniform time measure, growth in size is more or less linear. Indicated by arrows around the linear trend line is a visible ripple. This cyclic effect can be understood if the total organization is viewed as a self-stabilizing feedback system. That is, the design-programming-distribution-usage system has a feedback-driven and controlled transfer function and input-output relationship.

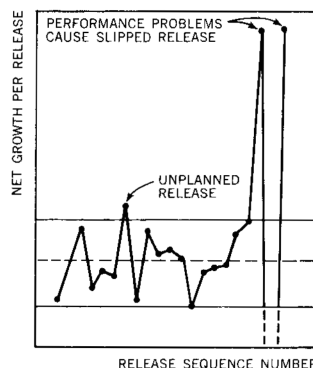
Figure 5 History of growth in number of modules. C: consolidation effect; F: fission effect



Some feedback results, for example, from constant pressure to supplement system capability and power. As the growth rate and work pressures build up, thereby increasing the size and complexity of the operating system, reduced quality of design, coding and testing, lagging documentation, and other factors emerge to counter the increasing growth rate. Sooner or later, as indicated by the segments marked C, these lead, at best, to a need for a system consolidation, a release that contains little or no functional enhancement and in which correction, restructuring, and rewriting activities predominate. As a result the system size does not grow significantly during such a release and may even shrink. At the worst a fission effect F may occur, as at $RSN = 20$ to 21 where excessive prior growth has apparently led to a break up of the system.

Figure 6 presents the net growth of OS/360 in each release. Analysis confirms the cyclicity of the growth process as indicated in the figure. A second observation may, however, be of even greater significance in estimating the limits of growth. With three exceptions, the net growth points may be seen to lie in a band bounded at about the 400-module level, a level that does not appear to have changed significantly in size during the lifetime of the system. Moreover, in the three instances where this growth level of OS/360 was exceeded, the record shows that, in the first case, the release was of such quality that it had to be followed by an unplanned clean-up release. The later two cases had equally unplanned consequences, significant schedule slippages, relatively disappointing performance, and—in the case of release 20—the previously unplanned division of the operating system into at least two independent systems. Moreover, note that releases with net growth near or in excess of the indicated bound tend to be followed by one or more releases with a much reduced net growth.

Figure 6 Cyclic nature of net growth of operating system releases



If we may generalize our conclusion, it is that as a large system grows through the addition of new and modified code, the system requires the regular establishment of a unique base reference to both code and documentation, such as is attained when the system is to be released for significant usage outside the development and maintenance group.

Also, in the present state of the art, complete and unambiguous specifications of changes or additions to be made are not normally achieved or even achievable. Nor is it possible to continuously prove the specifications to be consistent, and their subsequent implementation to be correct with respect to the new program behavior desired (or even with respect to previous program behavior). Hence the code and the system are tested. But tests can reveal only deviations from desired or expected behavior,[7] they do not demonstrate absolutely correct behavior or the absence of faults. Furthermore, the extent to which testing reveals deviations or faults is limited by both the resources that can be consumed to conduct them and by the view that test designers and interpreters have of the total program, the changes, and the intended behavior of both.

Thus, a further intrinsic consequence of system release is that the program is suddenly exposed to an environment in which both the expected behavior and the actual usage may—and usually do—differ from that to which the system was exposed in the development, maintenance, and test environments. Inevitably, therefore, release of the code results in the discovery of new faults. We conclude that sufficiently early release to users of stabilized code and documentation prevents a build-up of undiscovered faults. On the other hand, too many code changes that are undertaken without exposure to a wider usage pattern than can be generated in any test shop causes an accumulation of interrelated faults and system weaknesses, such as poor performance, that are far more complex to unravel. The data on which Figure 6 is based suggest that there existed a nonlinear effect with a critical growth mass in the operating system we are discussing of some four hundred modules.

This critical growth mass had been essentially invariant in almost a decade of OS/360 project and system life, despite methodological and technological improvements: increasing use of high-level languages and programming support tools; and increasing experience of designers, implementers, and management. Thus the characteristic is likely to be an attribute of the entire organization that relates to this system. That is, we appear to have identified a combined system and metasystem invariance. In view of the posited multiloop feedback nature of the process, one can expect to change and improve this characteristic growth rate only when one begins to understand the structure of the process and its relationship to the organization and to the system.

Without speculating further about the nature of the process, we may represent its invariance as observed in the present data by the following relationship:

$$\Delta M_R = K_{11} + S_1' + Z_1' \quad (1)$$

or by

$$M_R = K_{10} + K_{11}\, R + S_1 + Z_1 \quad (2)$$

Here ΔM_R represents the net growth of the system between $(RSN) = (R - 1)$ and $(RSN) = R$. A least-squares fit to the available data yields values of 760 and 200 for K_{10} and K_{11}, respectively. The S and Z terms represent the cyclic and stochastic components whose nature and magnitude can be determined using statistical techniques, such as those described in Reference 8. The small number of available data points, however, restricts the possible significance. We note that Equations 1 and 2 reflect directly the First and Third Laws proposed in the introduction of this paper.

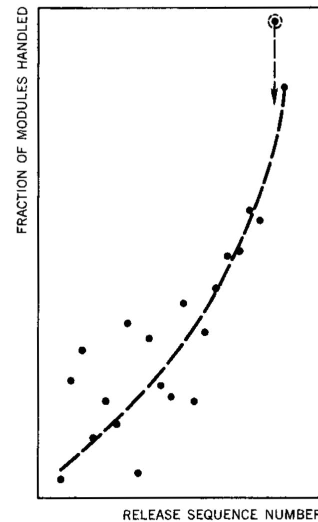


Figure 7 Complexity growth during the interval prior to each release

In the absence of a more satisfactory measure, we represent the complexity of the activity required during the interval preceding release R by the fraction C_R (of modules of the total system) handled. Figure 7 shows this measure plotted against RSN.

One possible (and least square-wise significant) fit is by a quadratic in R. Other functional forms (particularly an exponential fit) are also significant. Both the quadratic and exponential representations appeal to our need for models and limitations on the program development process, but more data will have to be obtained to determine the one that more closely reflects a particular process. On the basis of the principle of parsimony,[8] we select the following quadratic form for the current model:

$$C_R = K_{20} + K_{21}\, R + K_{22}\, R^2 + S_2 + Z_2 \quad (3)$$

For the present data, K_{20}, K_{21}, and K_{22} are respectively 0.14, 0, and 0.0012.

We note immediately that the monotonic growth trend implied by Equation 3 supports the Second of our three Laws. The Third Law is once again supported by the identification of a significant trend.

Notice that the residuals for this quadratic fit, and equally those for an exponential fit, are generally rather large for $R = 2$ through, say, $R = 14$. This variation is, of course, absorbed by the cyclic and stochastic terms, but in fact the residuals correlate very strongly with the handle rate HR_R. This correlation is not statistically conclusive, since both measures are in the present instance derived from related parameters. Nevertheless, it suggests a more complete representation of the following form:

$$C_R = K_{20}' + K_{21}'\, R + K_{22}'\, R^2 + K_{23}'\, HR_R + S_2' + Z_2' \quad (4)$$

where a least squares fit to the present data yields the values 0.037, 0, 0.0013, and 0.008 respectively, for coefficients K'_{20}, K'_{21}, K'_{22}, and K'_{23}.

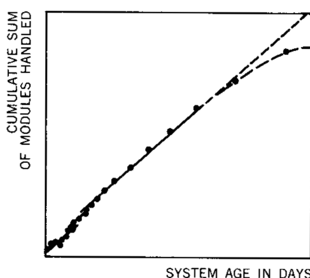
An interpretation of this model suggests that more rapid work leads to greater pressures on the team, and hence to more errors—which, in turn, require greater repair activity. The data indicate that this is mainly incurred in the same release rather than discovered and undertaken thereafter. Furthermore, since it appears to lead to an increase in the fraction of the system handled, it suggests that the maintenance teams tend to remove the symptoms of a fault rather than to locate and repair its cause. This deduction has been confirmed independently by a number of observers of—and participants in—the process, a fact that strengthens one's confidence in Equation 4 as a more complete representation of one aspect of the process.

work rate

The work associated with each release is measured in this instance by modules handled MH_R. This measure is, in each case, associated with a particular release and also with the release interval that separates the release from its predecessor. However, many releases overlap—particularly those releases that include major functional growth—and a new release may be integrated successively against two or even more predecessor releases.

Data on the degree of overlap between the various releases were not available to us. Therefore, we first examine the cumulative sum of modules handled (CMH) as compared with the age of the system, in an attempt to neutralize the overlap effect in determining the handle rate. Figure 8 shows these data fitted, as a first approximation, by a straight line. Such a fit suggests that the major changes that have occurred during the lifetime of the operating system in methodology, tooling, and staffing levels have had no significant impact on handle rate. This has stayed essentially constant over the period at some eleven modules per day.

Figure 8 Handle rate of modules over system lifetime

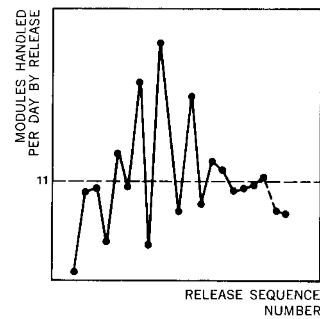


The data at the extremes of Figure 8 suggest that in the early life of the system, and in the most recent two releases, the handle rate may have been a little lower. This can no longer be confirmed for the older data. As far as present trends are concerned, however, since the handle fraction is approaching unity, we expect the scope of the cumulative handle plot versus system age to drop off from its previously constant value. It appears that even though the straight line fit is adopted as an initial model, an S-curve provides a more faithful representation over the life to date of the operating system.

We may now usefully examine the handle rate HR_R as determined by the ratio of the handle-to-release interval for each release, as shown in Figure 9. Because of the effect of release over-

lap, the range of rates achieved is exaggerated, but it is indeed centered around an average of about eleven modules per day. Also note that where the release rate has exceeded this average, the figure for the next release is lower. We conclude from the data for Figures 8 and 9 that the handle rate is stationary with cyclic and stochastic components that are confirmed by analysis to be significant and to have a three-release cycle.

Figure 9 Handle rate as a function of release number



Thus we adopt as our third relationship an expression of the following form:

$$HR_R = K'_{31} + S'_3 + Z'_3 \tag{5}$$

or

$$CMH_D = K_{30} + K_{31}\, D + S_3 + Z_3 \tag{6}$$

size as a function of age

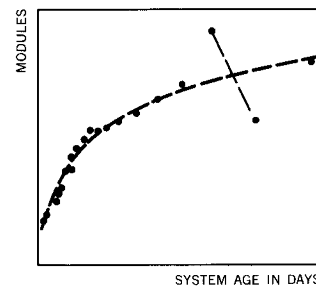
CMH_D counts the total number of modules handled between the first release of the system and day D, that is, when its age from release 1 is D days. HR_R represents the module handle rate in the Rth release interval. The S and Z terms once again represent the cyclic and stochastic components. For the present system, K_{30} and K_{31} are 1100 and 11 respectively. The statistically significant determination of a long-range trend with cyclic and stochastic components once again confirms the proposed Third Law.

We must now consider the data of Figure 5 which we have presented as a function of real time in Figure 10, where system size in modules is plotted as a function of system age in days. As indicated earlier in this paper, the relationship developed to represent this trend must be compatible with those already expressed in Equations 1 through 6. Of the alternative forms that can be significantly fitted we have selected the following expression:

$$M_D = K_{40} + K_{41} \log\,(1 + D/K_{42}) + S_4 + Z_4 \tag{7}$$

Here, a least squares fit yields K_{40}, K_{41}, and K_{42} as 89, 1350, and 51 respectively. The value of the intercept is not significant because the representation is not meaningful where D approaches zero. In reality, of course, system age was not zero at the time of $R = 1$, which is the assumed origin of our time scale. Nor, in view of the assumption that the build and maintenance processes are intrinsically different, may we expect to express the actual system age at first release in the same terms, even if this were known.

Figure 10 System size as an indication of declining growth rate



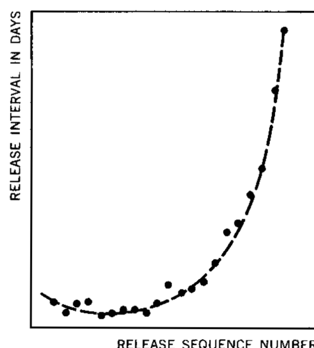
We note that the logarithmic representation is not asymptotic. Nevertheless, it suggests unlimited growth potential, though at a decreasing rate. This corresponds to our intuitive understanding that, as a system ages, it is always possible to change another instruction or add another module. However, the time required to do this tends to increase, unless the system is restructured and cleaned up.

One further observation of interest follows from the logarithmic representation selected. This representation is compatible with the constant incremental growth implied by Equation 1, provided that the release interval is growing polynomially, or, in the limit, exponentially. But this is precisely the behavior of interval growth, as shown in Figure 11. As it so happened, the earliest and very successful forecasting undertaken by us was based on this very observation and on the resultant exponential fits to the data.

summary

Equations 1 through 7 provide a model of the maintenance process for the operating system, OS/360, based on five parametric concepts, but with only four available measures. The model would be complete with the determination of the statistical parameters of the cyclic and stochastic terms. The small number of data points, however, precludes the determination of significant values.

Figure 11 Increasing release interval



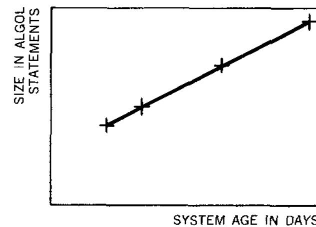
Recognizing the essential interdependence of the various parameters, one can also gain in descriptive power by determining compatible multivariate relationships such as are shown in Equation 4. These relationships could, of course, involve additional or lower-level breakdowns of existing parameters.

The number of basic relationships presented has been deliberately restricted to the number that is necessary and sufficient with respect to the existing degrees of freedom. Equations (1) through (4) have been selected because they bring out apparent invariants of the process. The recognition of invariances is fundamental to the application of the scientific method. As such, invariant detection in an analysis of the programming process not only strengthens our basic assumption of regularity in the process development, but it also provides hope that the analysis can be further developed and eventually permit improvement of the process.

Although the present model represents the observed behavior, it does, however, not explain it. Moreover, the representations break down at the extrema of observation. We have commented on this in the case of Equation 7 when D approaches zero from above. Similarly, Equations 3 through 6 are seen to be invalid representations as the fraction handled approaches its intrinsic limit of one. In fact, the expected nonlinear trend is visible in Figure 8. Good reasons have been given, however, for expecting a constant handle rate to be valid over the major portion of the interval considered. Thus it is not surprising that forecasting and planning techniques based on these representations have been useful in providing accurate data to improve planning in this particular environment.

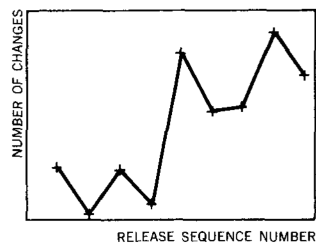
It now appears that further development of statistical process models should be directed toward an examination of the behavior of other systems from both the same and from other program development organizations, so as to determine the range of applicability of the observed phenomena. First confirmation has come from data on a second though smaller operating system that originated in the same organization. With minor differences, this operating system shows the same characteristics and trends, though with markedly different parameters. Preliminary data from a totally different organizational environment have also been examined[9]. As indicated in Figures 12 through 14, the smaller operating system confirms the basic observations of constant growth trends, cyclicity, overall smoothness, and declining work rates. The confirmation that this implies is of particular interest because the source is a programming organization outside IBM that created structured programs in ALGOL for IBM use only. Thus the organizational environment is quite different, but the phenomena are visibly present.

Figure 12 Constant growth rate exhibited by a second operating system



Clearly, these data—especially the invariants—should be studied further, for example by examining actual work rates within a release interval. With further study, one hopes to discover the reasons for the phenomena and ultimately to remove the limitations that they imply.

Figure 13 Number of changes as a function of release number of a second operating system

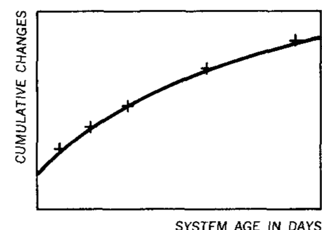


In parallel with the study of invariants, one should also proceed with the development of abstract models that represent and formalize our perception and understanding of the large-program development process itself or of aspects of the process. We describe examples of our earliest approaches to this problem in the following section.

Formal modeling of the program development process

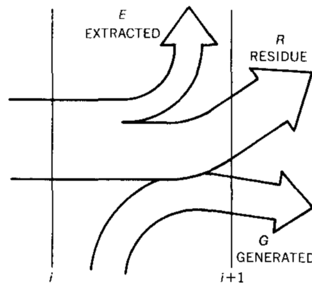
Since our goal is to understand and to learn to control the programming process, one view of the process is to see it as the interaction between two entities. On the one hand, the large program in all its representations and with its documentation we call the "object." On the other hand, the human organization that implements the process in its manipulation of the object is termed the "team." The function of the team is to execute changes in the object.

Figure 14 Declining work rate exhibited by a second operating system



In conjunction with user-provided data, the object enables a computing machine to perform useful work. During its lifetime, all kinds of changes to the object are necessary. The (hardware) machine, or some of its components may be changed or replaced. New devices may be added. Computing requirements may be redefined to serve new uses. New ways of using the sys-

Figure 15 Primitive model of fault penetration



tem may be devised. In general, the behavior of the system deviates from that anticipated or desired because of *faults* in the system. We term faults related to changes in the environment *defects*, whereas an *error* relates to the difference between actual and anticipated behavior. When faults manifest themselves, the team is required to undertake corrective action, to perform changes on the object.

Observations related to those discussed in the previous section suggest that system evolution is to some considerable extent influenced by fault repair activity. Our earliest formal models, therefore, have been designed to examine fault distribution in the system. These models were based on the following assumptions:

- Changes, that is, object handlings, are, in general, imperfect. When changes are performed, errors are injected by the team with probability greater than zero. This by itself would imply a continuous need for change, even if the environment were fixed.
- There is a delay between the injection of an error and its first detection and recording, and another delay exists between recording the error and its final elimination.
- Some errors are ordered in that one of them must be repaired before the other can be detected. That is, there is a layering of errors in the object that is representable by a directed graph.
- The team creates and uses documents, which are kinds of representations of the object, to study faults and possible courses of action. The documentation may be viewed as an integral part of the object.
- Team members, while involved with changes, communicate with each other in the lanugage of these documents.
- Team members have to be educated in the documentation; moreover, the team has the additional task of updating the documentation to reflect changes performed on the remainder of the object.
- Deficiencies in documentation influence the effectiveness of the process and, therefore, cause deficiencies in the object.

From these assumptions, we have developed two classes of models. The first emphasizes the internal distribution and propagation of errors in the object. The role of the team is simply to eliminate observed faults.

The second class of models gives the team a more active role. Management is free to make decisions as to those particular

tasks, error repair, documentation, or other activities to which the team should turn. The object responds to these actions by manifesting different error generation rates.

model of fault penetration

The model of fault penetration that we now discuss is a measure of complexity due to aging. Consider an elementary change activity in the time interval $(i, i+1)$. This is depicted in Figure 15, where the width of each arrow band may be interpreted to be proportional to the number of faults it represents.

At time i, a number of faults is assumed to exist. As a result of team activities, the following occurrences are likely:

- A fraction E of the total faults is removed (extracted).
- New faults G are injected (generated) due to imperfection in the activity.

Thus at time $i+1$ a new composition of faults appears that consists of residual R and generated errors.

Preserving the distinction between residual and newly generated errors is fundamental to an understanding of the evolutionary process. A system cannot be effectively maintained if that distinction is not understood. And complete understanding demands a knowledge of the history as well as of the state of the object at all times.

The primitive change activity of Figure 15 spans the network of Figure 16, where i is a discrete measure of age (or release number) and j is a variable used to introduce the tree structure. For each node, the residual design faults R and the generated faults G may be expressed as follows:

$$R_{i-1,j} = R_{i,2j-1} + E_{i,2j-1} \quad (8)$$

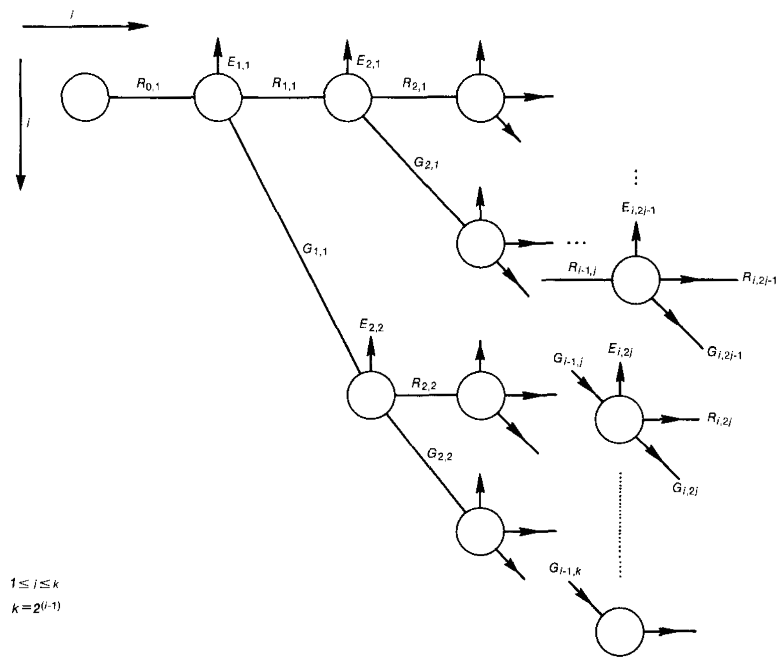
or

$$G_{i-1,j} = R_{i,2j} + E_{i,2j} \quad (9)$$

and $G_{i,2j-1}$ and $G_{i,2j}$ are to be defined for each node by the following additional assumptions:

- $G_{i,j} > 0$ (imperfection hypothesis)
- We define $C_i = 2^{i-1}$ *fault classes* for every i. Each class has a unique label that consists of a two-valued $\{R,G\}$ character string of length i, with the first element always R, meaning residual design faults. For example, $R\ R\ G\ R\ G\ G\ R$ represents a node or fault class at $i=7$. More specifically, faults in this class are the

Figure 16 Network showing faults extracted and faults generated



residue $(\ldots R)$ of
  faults generated $(\ldots GR)$
    while extracting faults generated $(\ldots GGR)$
      while extracting the residue $(\ldots RGGR)$ of
        faults generated $(..GRGGR)$
          while extracting the residue $(.RGRGGR)$ of
            faults in the original design $(RRGRGGR)$

The model as described represents an increasingly large and complex network of fault trajectories or histories, even though the total number of faults present may have been stable or even declining as a consequence of nonzero Es. Faults are identified in terms of unexpected or undesired system behavior in execution. Thus we have excluded from consideration here simple faults that manifest themselves locally in a single element of the system. That is, we may omit from consideration those faults that may be detected or removed by operating with or on any one element alone, and consider those situations where rectification of a fault requires coordinated changes in two or more system elements and in their interfaces. Interactions among interelement and intergeneration effects represent the conceptual complexity of the fault pattern. And it is the increasingly complex fault structure that underlies increasing object complexity. Thus

periodic restructuring of the object is necessary to reduce complexity because increasing object complexity is itself a fault that impinges on the maintainability of the system.

The connection between the relational complexity of errors and the structural complexity of the system implies that relational complexity may be a measure of communication requirements for the team and the underlying cause of fault extraction and generation over the entire lifetime of the object.

We now give a quantitative interpretation of the fault penetration model, which is a simplified view of structural aging. To analyze the above fault generation model so as to obtain even a simplified view of the resultant structural aging, additional assumptions must be made about the fault extraction and generation variables E and G.

The simplest hypothesis is that, for each node,

$$E = G \tag{10}$$

that is, as many faults are extracted as are generated. Under this assumption the system appears to be in a steady state.

Let us consider the number of fault classes C_i as a measure of *complexity* of the system. Analysis has shown that complexity increases even in steady state, that is, when the number of faults in the system remains constant.

A degree of freedom can be eliminated by establishing a relation between fault extraction and the fault content of a given class. A reasonable assumption could be that $E_{i,2j-1}$ is in fixed proportion to $R_{i-1,j}$, and no new errors are generated. Thus we have the following fault elimination-to-fault residue ratio:

$$\frac{E_{i,2j-1}}{R_{i-1,j}} = (1 - p) \tag{11}$$

and fault decay follows a geometric distribution with parameter p, which is constant such that $0 < p < 1$. After i intervals, and having started with a given collection of faults S, the remaining number of faults in the original collection is $S(1-p)^i$, whereas $S(1 - (1-p)^i)$ faults must have been extracted. Since $G_i \equiv 0$ for all nodes, the system approaches an error-free state asymptotically (approximately exponentially). Thus in all cases considered the geometric distribution reflects the reasonable assumption that the smaller the fault content the fewer the faults there are to be discovered and extracted. This, however, still implies a monotonically increasing C_i until, if ever, a fault-free state is reached.

More elaborate relations between E and G may be required, so as to represent currently observed situations. It is important to note that E and G at each node are not independent, but are coupled via the team and the process.

qualitative interpretation of fault penetration

As already indicated, even with decreasing fault content ($E > G$), the complexity measure C increases monotonically. This results from and reflects the increasing stratification of the system because of the increasing heterogeneity of faults.

The resultant structural deterioration experienced as an increasing difficulty in executing change alerts the team to the need to counteract the aging process. On the basis of our previous assumptions, the latter may be considered proportional to $2^{G(i)}$, where $G(i)$ is a monotonically increasing function of i that reflects higher-order variations not considered here, as well as the complex relationship between fault and system structure.

To cope with the situation, the state of the system has to be precisely defined. Documentation must be accurate, complete and accessible. In addition, the administrative organization or responsibility of team members must be well defined. Finally, team members must be aware of the state of the system by learning. Fulfillment of these needs can effectively reduce the effect of growing complexity, and can be represented symbolically as follows:

$$C_i \text{ (modified)} = \frac{2^{G(i)}}{2^{DAL(i)}} = 2^{G(i)-DAL(i)} \qquad (12)$$

where DAL means "Documentation, Accessability, and Learning," which are constructive factors. Equation 12 is a qualitative one, and one that is closely related to our earlier fault penetration model. Real-life situations are much more complex. Communication complexity required to overcome system stratification may, for example, be further increased by geographic scattering of the team activity. Nevertheless, the model enables one to address some very real questions about the program maintenance process. For example, since the model mirrors a domain that is discrete (indexed with i), the model suggests that perhaps increasing the number of intervals i (i.e., decreasing the inter-release time) should permit faster extraction of faults. This would occur if such an increase were to imply an increase in the frequency of restructuring and of providing adequate team knowledge of the state of the system. That is, $G(i)$ and $DAL(i)$ must be kept in step. Whether more frequent intervals would indeed be beneficial is by no means clear. As a consequence of one of our early assumptions, namely, that faults are layered and

manifest themselves in a partially ordered fashion, one has to go through the process of gradually repairing the system, with the inevitable result of generating complexity. In addition, short intervals provide less opportunity to exercise the system in actual use for fault manifestation, thus reducing the number of faults that can be extracted. The size of the optimum interval is, therefore, undecided. A more detailed model is required if this is to be formally explored with the objective of helping solve a problem that arises in real system development.

management decision model

We now discuss our management decision model, which reflects our earlier formulation,[4,10,11] and which is based on the following assumptions.

Budget B, the available budget, bounds the total activity. During the change process, every unit of fault extraction (termed "progressive" P) activity, measured by $G(i)$ in the model given by Equation 12, is associated with a certain amount of documentation, administration, communication, and learning activity (termed "antiregressive" A) as measured by $DAL(i)$ in Equation 12.

Neglect of A activity results in the accumulation of additional work demand to cope with increasing complexity C. This cumulative demand can be removed only by a (temporary) increase in the intensity of A, which, as a result of the limited budget B, causes a (temporary) decrease in progressive activity P.

Management is assumed to have full control of the allocation of its resources and the division of effort between P- and A-type activities. Management cannot, however, directly control the growth in complexity that accumulates, except by utter concentration on complexity control through restructuring. This is an activity that is strictly antiregressive and, as such, is psychologically difficult to inspire, since it yields no direct, short-term, benefits.

To examine these concepts further, we now present an alternative formulation of the model. In a somewhat simplified fashion, we assume that resources are fixed (by budget) and that they are equally applicable to either P or A activity. B and activities P, A, and C can be measured in cost per unit of time, which express the budget rate and its expenditure rate on progressive, antiregressive, and complex control activities, respectively. In addition, we use the following relationships:

$k = A/P$ represents the inherent A activity required for each unit of P activity, so that complexity does not grow.

m = management factor, which is the fraction of progress kP that is actually dedicated by management to A activity.

At any time, the total expenditure on all activities must be equal to the budget, hence the formula for the budget is given as follows

$$B = P + A + C \tag{13}$$

The formula for antiregressive activities is

$$A = mkP \tag{14}$$

and

$$C_A = \int_0^t (1 - m) kPdt \tag{15}$$

where

$$C_A(t_0) = 0$$

The expression C_A or complexity reflects the cumulative decay caused by the neglect of A activity.

Since the values k and m are left free to vary with time, the model can be used for the investigation of the consequences of various possible management strategies in controlling the maintenance process. Further freedom can be introduced by inserting variable-length delays among the three major expenditure components. A large problem space thus results that can be explored by interactive modeling for increased insight. In this environment, real-life observed phenomena can be approached in the model by stepwise changes in model parameters.

management simulation

A graphic modeling facility has been used by the authors. This system was essentially an analog computer that was implemented on a digital machine such that the analog components (delays, adders, integrators, etc.) could be connected into a network on a cathode ray tube by the use of a lightpen. Upon request, the computer accepted the network and numerical parameters as inputs for a stored program. The system then computed the response, as described in References 12 and 13.

During the numerous experimental sessions with this facility, many real-life phenomena were successfully reproduced. One example was the cyclic pattern of object growth for the statistical model discussed earlier. The network consisted of a nested two-loop feedback system; preset threshold values for k simulated the management decisions.

More precisely, in our simulation, after a period of persistent neglect of A-activities ($m < 1$), management becomes alarmed by the rapid reduction of P due to increasing C. Consequently, an increase in A is scheduled ($m > 1$) until the situation notice-

ably improves. At this point, management again becomes optimistic and relaxes k to a lower level. In the long run, however, C grows monotonically. A sample output of a run is presented in Figure 17.

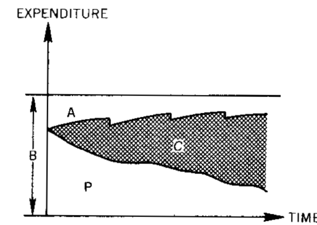


Figure 17 Example output of a budgeting simulation

The authors are convinced that this type of interactive modeling is perhaps the most fertile, and certainly the fastest, way of developing a feel for the interactions involved, and gradually developing a more complex model that has the power of predicting real-life behavior.

In contrast to previous models, management decision modeling yields an optimistic prognosis, since it includes parameters that reflect management discretion. Thus it permits the counteraction to remove the consequences of growing complexity, action that occurs in real-life situations. On the other hand, of course, the model does not reflect the internal structure of the object. In our earlier models, internal structure was modeled by combining the management model with an extension of the fault penetration model.

model of limited growth

Suppose that management is free to allocate resources to grow the object, as well as to extract faults as in the previous model. Of course, both activity classes are essentially imperfect in that, while performing them, errors are injected into the object.

As the simplest case, we would like to show how the size m of the object, measured, for example, by the number of modules it contains, develops in the presence of error generation that is proportional to growth activity. In signal-flow-graph form, the linear relations can be represented by Figure 18. Here E and R convert growth rate and error repair to work demand (measured in man-hours). F is the error generation rate (the number of errors per man hour) and r is the number of errors.

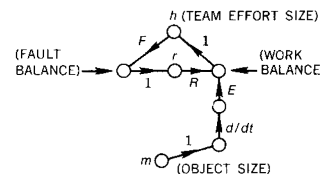


Figure 18 Model of growth in the presence of error generation that is proportional to effort

The corresponding equations are:

$$h = Rr + E\,dm/dt \tag{16}$$

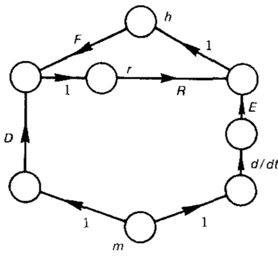
$$r = Fh \tag{17}$$

Assuming a constant work force h, the solution is given as follows:

$$m = m_0 + \frac{1 - RF}{E}\,ht \tag{18}$$

where growth is a linear function of time. The greater the work force and the smaller the error generation, the more rapid is the growth, which is, in principle, unlimited. The reason is that, on the basis of our previous assumptions, the effort not used for repair is available to grow the object at a rate that is independent of its size.

Figure 19 Model of limited growth in which increasing size generates increasing errors



Observations on our previous models, however, have suggested that larger and older objects are more complex and receive more errors as they evolve—through growth and through fault removal. Retaining the linear character of the relationships, the flowgraph given in Figure 19 represents the modified assumption, namely, that increasing size causes more errors to be generated, with a gain D per unit size. The somewhat modified equations appear as follows:

$$h = Rr + E\,dm/dt \tag{19}$$

and

$$r = Fh + Dm \tag{20}$$

where Equation 19 represents a negative feedback to control size. The solution now becomes the following:

$$m = m_{cr}\left(1 - \frac{1}{m_{cr}}\, e^{-\frac{RD}{E}t}\right) \tag{21}$$

$$m_{cr} = \frac{h(1 - FR)}{RD} \tag{22}$$

Equations 19 through 22 indicate that under the assumptions of this section growth is limited to m_{cr}. This critical size can only be reached asymptotically. The reader may be wise to compare this result with the real-life observations previously reported.

The critical size can be increased by increasing the size of the work force. However, this means that a subsequent reduction of the work force can create a new critical size that is smaller than the one already reached. Thus a situation of monotonically increasing error content is created.

This model has been studied under differing assumptions. The main conclusion remains, however, that object size is limited with even the slightest negative feedback of size.

This section has presented several models each throwing different, though related, light on the program maintenance or enhancement process. Our aim has not been to present completed models. Rather, we have wished to illustrate how the modeling may be approached, and how interpretation of the models may be used to study and to improve the programming process.

Concluding remarks

Currently, the process of large-scale program development and maintenance appears to be unpredictable; its costs are high and its output is a fragile product. Clearly, one should try to reach beyond understanding and attempt to change the process for the better.

As a first step toward ultimate improvement, we are studying the process as it is, and as it is evolving, much as the physicist studies nature. Out immediate goal is an organized quantized record of observations that formalizes the perception of what is happening and what is being done. With such global studies, one may hope to identify specific points or sources of trouble and perhaps identify areas of the process that are major causes of concern. When that which is happening is understood in the context of the process as a whole, one may attempt to understand why it is happening. Only then will one attempt to change the process without risking local optimization that is very likely to reduce significantly the degree of global optimization. At the present time, for example, it is not clear to what extent improvements should be sought by attention to the human organization, management, or by emphasis on the product side of the process, in order to achieve the most significant gain in and from the process.

We do speculate that communication is a major problem. If this can be confirmed then, for example, a design methodology that expresses the understanding and intention of the designer unambiguously and completely might eliminate many difficulties. One may also hope to avoid problems in the performance area as a consequence of overspecification. Thus one might equally consider that a reduction in product complexity, by better partitioning, for example, could lessen the need for communication and, at the same time, improve performance potential. To do this effectively, however, we must be able to identify those parts of the product that are most interlaced in their logical structure.

Our data so far have been largely limited to that of a few rather large operating systems that were produced within the same large administrative organization. Even these data are meager. Since the initial design phase, no one anticipated the long series changes that was to follow the initial development. We now know much better and are able to specify the kinds of data that are necessary for future analysis of the development, implementation, and maintenance processes.

We are also enlarging our scope beyond the environments studied so far. It is already clear that qualitative observations similar to ours have been made at other places where large-scale programming has been undertaken. This suggests an urgent need for the definition and standardization of process measures to facilitate meaningful comparisons between dissimilar systems, processes, and organizations.

Clearly, we still must test the generality of the hypotheses presented in this paper. It will, for example, be of major interest to determine the degree of generality and the range of validity of

the various invariants discovered so far, after filtering new data to remove noise due to environmental factors. This should improve the usability of program evolution dynamics concepts and techniques as planning tools, an improvement much needed by managers who are, in general, not very successful in assessing, predicting, and controlling schedules and resources in the software process.

It is important for an emerging discipline, such as program evolution dynamics, to summarize its most essential concepts into unambiguously defined and measurable quantities at an early stage in its development. This makes it possible to use appropriate techniques and tools from established disciplines. Mathematics, for example, facilitates comparisons between derived results and real life, and may even help the development and communication of new ideas.

One of the most frequently used—but as yet undefined—concepts encountered in our studies is that of complexity. Particular definitions that have been established in the somewhat narrow content of computational magnitude do not appear to be useful or applicable for the study of structure and interaction. After some preliminary studies, we have concluded that a measure of complexity, applicable to the large scale programming environment, could be developed by using established concepts that are related to information, uncertainty, and entropy. Further investigation in this direction forms an ongoing activity in the authors' groups.

Given a measure of complexity expressed in terms of simple structural properties—such as the number of interactions between product or organizational elements—normalized measures for programming effort, productivity, system reliability, and security can be derived and comparisons between different products or methodologies made meaningful. Without such a measure, many of the essential parts of the developing discipline remain unconnected and phenomena are easily misunderstood. An early result in the study, for example, suggests the consideration of complexity of software and its documentation in a unified fashion. In this case, the total project workload can be better quantified, and plans and schedules made more accurate, provided that the manpower need is strongly related to complexity.

Many of the directions pursued in our exploration of evolution dynamics appear to relate to the global properties of complex systems rather than to properties that result specifically from the software environment.

Thus we assume that the results of our studies may be generalizable to other complex technological projects, and to the study of sociological, economic, and biological systems or organisms. In the immediate future, however, we shall concentrate our studies on the evolution of large programs, since in this area change is observable over a relatively short period of time, and experimentation is possible without the serious penalties that could be incurred in other fields. Thus program evolution dynamics may be interpreted as a suitable prototype or test bed for the study of more general system evolution dynamics.

ACKNOWLEDGMENT

The authors appreciate the contributions of their many colleagues in IBM and in the Imperial College, and in particular their discussions with Heinz Beilner and Lip Lim. The CSMP modeling was a contribution made by Steve Morse.

CITED REFERENCES

1. E. W. Dijkstra, "Notes on structured programming," pp. 1-82 O. J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, *Structured Programming*, Academic Press, New York, New York (1972).
2. E. W. Dijkstra, "A constructive approach to the problem of program correctness," *BIT* **8**, 174-186 (1968).
3. M. M. Lehman, *The Programming Process*, IBM Research Report RC 2722, (December 5, 1969), IBM Thomas J. Watson Research Center, Yorktown Heights, New York 10598.
4. M. M. Lehman, *Programs, Cities and Students—Limits to Growth*? Inaugural Lecture, Imperial College of Science and Technology (University of London) London, England, May 14, 1974.
5. L. A. Belady and M. M. Lehman, *Programming Systems Dynamics, or the Meta-Dynamics of Systems in Maintenance and Growth*, IBM Research Report RC 3546 (September 1971), IBM Thomas J. Watson Research Center, Yorktown Heights, New York 10598.
6. D. R. Cox and P. A. W. Lewis, *The Statistical Analysis of Series of Events*, John Wiley and Sons, New York, New York (1966).
7. E. W. Dijkstra, "The humble programmer," *Communications of the ACM* **15**, 859-866, (October 1972).
8. G. E. P. Box and G. M. Jenkins, *Time Series Analysis*, Holden-Day, Inc., 500 Sansome St., San Francisco, California 94111.
9. D. H. Hooton, *A Case Study in Evolution Dynamics*, M.Sc. thesis, Department of Computing and Control, Imperial College of Science and Technology (University of London), London, England (September 1975).
10. L. A. Belady and M. M. Lehman, "An introduction to growth dynamics", *Statistical Computer Performance Evaluation*, Academic Press, New York, New York (1972).
11. L. A. Belady and M. M. Lehman, *A Systems Viewpoint of Programming Projects*, Imperial College Research Report 72/31, Imperial College of Science and Technology (University of London), London, England (1972).

12. H. B. Baskin and S. P. Morse, "A multilevel modeling structure for interactive graphic design", *IBM Systems Journal* 7, 3–4, 218–228 (1968).
13. *1130 Continuous System Modeling Program*, Order No. H20-0209-1, IBM Data Processing Division, White Plains, N.Y., 10504.

Reprint Order No. G321-5035.