DETERMINISTIC AND STOCHASTIC SOFTWARE LIFECYCLE MODELS AND CHAOS

H.M Hubey, Assistant Professor
Department of Mathematics and Computer Science
Montclair State College
Upper Montclair, New Jersey 07043
hubey@apollo.montclair.edu
201-279-1603 Home
201-893-4000, x5269, x5132, x4263

ABSTRACT

This paper explores the software development effort with the development of several differential equation models. These models are one parameter models of the type in Parr and Putnam papers. The models due to Putnam and Parr model the software development efforts at almost the extreme ends of the spectrum in terms of difficulty. This paper explores the software development effort with several differential equation models. In these models whether the modeler incorporates the planning and specification stages into the model is left up to the modeler as a "matter of taste." Additionally, the "inherent difficulty" of the software development project is taken into account with the coefficients of the models. The models deal with the time evolution behavior of a parameter which represents the "essential completeness" of a software project which might be interpreted in the sense that it is complete enough to be put into operational use. All the models are shown to have the property that the "completeness" converges to unity as time increases to infinity.

The stochastic models in the paper treat the case in which "completeness" of a project can be interpreted to mean the "absence of bugs". Two stochastic models are developed. It is shown that the mean of the model driven by Gaussian white noise converges to unity as time goes to infinity. The second model is driven by nonwhite one-sided noise to make it more realistic. The Fokker-Planck equation for this model is derived and the stationary Characteristic Function of the stochastic process is obtained. It is shown that with this more realistic model the project can only be completely free of bugs only with infinite resources and infinite time.

Much more interesting results follow if time is modeled as a discrete parameter. This discrete variable , one-parameter model is shown to possess chaotic properties --which sheds light on the difficulties of large-scale software development projects.

DETERMINISTIC AND STOCHASTIC SOFTWARE LIFECYCLE MODELS AND CHAOS

H.M Hubey, Assistant Professor
Department of Mathematics and Computer Science
Montclair State College
Upper Montclair, New Jersey
hubey@apollo.montclair.edu

I. INTRODUCTION: SOFTWARE DEVELOPMENT FACTORS

This paper explores the software development effort with the development of several differential equation models. These models are one parameter models of the type in Parr {20} and Putnam {22} papers. The models deal with the time evolution behavior of a parameter which represents the "essential completeness" of a software project which might be interpreted in the sense that it is complete enough to be put into operational use. There are other interpretations possible. "Completeness" can be defined as the total or the probabilistic absence of bugs. For software projects which are not part of "embedded" systems or the like where a glitch can have catastrophic consequences, the first definition of completeness is sufficient. The stochastic models in the paper attempt to treat the second case. Since most of the models in this paper (and in other one parameter models) have at best a few coefficients, the many factors that influence the cost, timeliness and reliability of software must be accounted for with only these coefficients. Any model building should start with a verbal description of the domain of model parameters. The major factors that influence software cost

as listed in Fairley (11) are:

Table I

Programmer Ability
Required Reliability
Level of Technology
Product Complexity
Product Size
Available Time

Programmer Ability: According to experiments by Sackman (24) as reported in CACM and discussed in Fairley (11):

> The differences between the best and worst performances were factors of 6 to 1 in program size, 8 to 1 in execution, 9 to 1 in development time, 18 to 1 in coding time and 28 to 1 in debugging time. Eliminating the extreme cases still resulted in a typical variation in programmer productivity of 5 to 1.

Required Reliability: According to Boehm{2} , the stricter the reliability requirements for a program, greater the effort (Boehm's usage of effort corresponds to programmer-months and is different ~~than~~ from Putnam's) that its development requires. The effort multipliers range from 0.75 for very low reliability requirements to 1.40 for very high reliability requirements.

Level of Technology: Boehm {2} has developed effort multipliers for software practices that take into account the language being used, and the software development tools being provided the programmers. According to Boehm, "the use of modern languages and practices can reduce development effort to 0.45 of that required using primitive tools and techniques". Using very modern technologies such as Fourth Generation Languages (4GLs) it would be possible to reduce the programming effort even more. In a study done by Rudolph {23} and reported in Martin {17}, the difference in elapsed

time between a project programmed in 4GL and COBOL was as much as seventeen to one.

Product Complexity: Some programs are inherently more difficult than others. This idea is explored in detail in Section III.

Product Size: It will be useful at this point to have some agreement as to what is meant by "large" or "small". The following categorization of program sizes is due to Yourdon (18) and is reported also in Fairley (8).

TABLE II

Category	# of Pgmrs	Duration	Product Size*
Trivial	1	1-4 weeks	500
Small	1	1-6 months	1K-2K
Medium	2-5	1-2 years	5K-50K
Large	5-20	2-3 years	50K-100K
VL	100-1000	4-5 years	1M
EL	2000-5000	5-10 years	1M-10M

VL= Very Large
EL= Extremely Large
* Source Lines

A seeming paradox associated with large programs is that neither the development time nor the manpower is a multiple of a small program--constant of proportionality being reflected in the relative sizes of the two programs. The reason for this is that the largeness itself adds to the product complexity (aside from the inherent (time) complexity of the program) in addition to problems associated with the communications problems as the number of programmers increases. Brooks was the first to note that the number of communication paths among programmers grows as n(n-1)/2. This idea is explicitly taken into account in the Putnam paper. In a time and motion study of programmers conducted by Bairdain{1} and as reported in

Fairley{11}, the proportion of time programmers spend in various activities is given below:

TABLE III

Writing Programs	13%
Reading Programs and Manuals	16%
Job Communication	32%
Personal	13%
Miscellaneous	15%
Training	6%
Mail	5%

The time spent in job communication is the largest category which would seem to vindicate Brooks. Additionally it can be seen that the time reading manuals and programs is significant and it would seem to lend credence to the necessity of "learning" associated with the project.

II. REVIEW OF PUTNAM AND PARR MODELS

Since the pioneering work done by Halstead {14} in what he termed "Software Science" many workers in the field have contributed to the efforts to model the software development cycle. There are many different models for the estimation of software development time and cost. An excellent and comprehensive review of this field is the earliest contribution and the "de facto standard" against which others compare their models is due to Putnam{22}. It is a linear differential equation model which is developed from what are considered to be fundamental and measurable parameters, such as the number of operators, the number of operands, the number of

unique operands and operators. It has also been argued by Comer and Halstead {7} that these paraaameters are the appropriate metrics for the study of top-down design of programming projects. The essential model, as ably summarized by Parr, is a differential equation of form:

1) $$dw/dt = q(t) [1- w]$$

where w is the work accomplished, and q(t) is the "pace" at which "work is being done" or "skill available" at time t which is brought to bear on the project (which is proportional to the amount of knowlege) and which, in the Putnam model, is proportional to the elapsed time t. Of course this assumption implies a "linear learning law". Actuallly Putnam's model is sightlly different and more complicated. In the model above, w has been normalized so that $0 \leq w \leq 1$, where a 1 (one) indicates that the project is complete. Parr has objected to the learning function asssociated with the project since it implies that the learning continues to increase after the project is completed{20}. The Parr model takes care of this seeming anomaly by making the learning linearly proportional to the propotion of the project completed. Thus he derives:

2) $$dw/dt = \beta w (1 - w)$$

In this model the βt term in the Putnam model has been replaced by a factor proprotional to w, the proportion of the project completed. The implication of this change is that the software being developed is such a new entity that the programmers' learning continues apace with the project itself. This equation is the well-known Riccati Equation and has been used to model population growth, spread of technological innovations (Braun[3]),

and also motion of bodies and growths observed in certain economic parametes {Davis[8]}. A side effect of this model is that the Parr model also implies that the manpower associated with the project at time zero is nonzero. Parr attributes this to the problems with the "formal accounting procedures for recording the amount of effort applied" and also suggests that the Putnam model is therefore incorrect. A comparison of the Putnam and Parr models can be found in Basili {21}. Figure 6.4 in Basili's paper is duplicated here for comparison purposes (See Figure IV). It seems to the author that it is a "matter of taste" whether one wants to consider feasibility studies ,requirements analysis and the like to be a part of the "software life cycle". The models presented in this paper are "flavored" accordingly.

III. ALTERNATIVE MODELS

It is well known in Production Theory that large scale, one-time projects are the most difficult to manage. An example would be a job shop or a one-time project such as a lunar landing. In contrast, continuous production systems which produce high-volume standardized products are the easiest. At a superficial level, it would seem that every software project is a one-time project with all the attendant problems. However some software projects such as small file-processing or database projects are more like continuous production systems in that the programmers' "learning experience" during the project's duration is very small compared to projects whose "novelty" might be large. For example, real-time software such as for process

control or software for "embedded systems" is more difficult to write. Not only do these systems have real-time constraints but they also have higher reliability requirements. Brooks {4} was the first to classify software projects along these lines. He divided programs roughly into three categories. "Program Production Software" (Utilities i.e. editors and compilers) is three times as hard to write as "Applications Software" and that "System Software" is three times as hard to write as Utility Programs. Boehm {2} uses a similar taxonomy and calls them "Organic", " Semidetached" and "Embedded". It seems to the author that in addition to the problems with "reliability requirements" and "space complexities, there is another problem that the difficult software projects share, namely, the "novelty factor"; that is, how much experience with this particular software do the programmers have. It seems reasonable for Brooks to have categorized Systems Software as the most difficult. Brooks wrote about the time when the field of Computer Science was not very developed. The fundamental programming concepts of Abstraction, Information Hiding, Structure, Coupling, Cohesion, and Modularity had not yet been developed, Operating System design concepts have now also stabilized to the point where there are now books available that go beyond the purely descriptive texts. (See for example Comer{6} and Tanenbaum{28}.) It also makes sense why the most difficult systems at this point in time are "embedded systems". These different types of programs require different learning curves. There is also no reason why software projects should be broken into only three categories. It is posssible then to develop other models

which may better describe the software development process.

Analysis of the Models:

It is clear that the two terms on the right hand side of equations are connected with the space and time complexities of the software project. In most software it is possible for programmers to make trade-offs between space and time complexities of the software project. For example, it would be possible to substitute table-lookups for computations. As the software project nears completion the complexity of the project increases and it becomes more difficult to add code to it. One might say that the "space complexity" of the project increases. This implies that the "rate of work" must decrease as the project progresses toward completion. So it is necessary for dw/dt to decrease. Hence, the (1-w) term in Eq.(1) is appropriate to reflect this observation. As for the "product complexity"; (from Fairley)-- actually what is usually called the "time complexity"; although it doesn't really change, the understanding of it by the programmer team increases as time passes. This observation then implies that the rate of "doing work" would increase as time passes. Hence the q(t) term should increase. It is only here that Putnam's model, Parr's model, and the models presented her differ. It is the assertion of this paper that q(t) should be appropriate to reflect the inherent complexity of the project. Specialized "first-time" projects , "embedded systems" and the like would have a different "learning curve" than typical file-processing type projects. Hence the modelers should select q(t) to faithfully reflect the project type.

The differential equation models considered in this paper are of the type shown in Eq (1), which can be rewritten as

3) $$\int dw/(1-w) = \int_0^t q(s)ds$$

Making the substitution $x = 1 - w$, we obtain

4) $$\int dx/x = -\int_0^t q(s)ds$$

Hence the solution is given by;

5) $$w(t) = 1 - (1 - w_0)\, e^{-\int_0^t q(s)ds}$$

Asymptotic reasoning. Too weak! What happens when $t < \infty$?

It can be seen that $w(\infty) = 1$ so that any equation of this type is a candidate for a Software Engineering model.

Example 1: This model is a slight generalization of the Putnam model since it is a slight alteration of the learning curve; i.e.

6a) $$q(t) = (\emptyset + \beta\, t^{\mu})$$

Putnam himself might have chosen this model had he intended a more general model for the software lifecycle. The solutionof the resulting DE gives

6b) $$w(t) = 1 - X_0 \exp\{ - \emptyset t - [\beta/(\mu+1)]\, t^{\mu+1}\}$$

where $X_0 = 1 - w_0$

It can be seen from the solution that w_o does not necessarily have to be zero so that this model can account for the preliminary work such as feasibility studies. However, the learning curve is still linear and the programmer team's knowledge and skills continue to increase long after the project is essentially complete. Some typical solutions are shown in Figure I. It should be noted that the solution, as a special case, reduces to the Putnam solution for $\mu=1$ and $\phi=0$. The rate at which work is being done is given by the derivative

$$7) \quad dw/dt = X_0 (\phi+\beta\, t^{\mu}) \exp \{- \phi t - [\beta/(\mu+1)]\, t^{\mu+1}$$

If $\mu = 0$ and $\beta = 0$, then we have another special case, one in which constant 'level of skill' is input into the software project. This model might be ideally suited for a typical trivial fileprocessing type project. In this case the solution reduces to

$$8) \quad w(t) = 1 - X_0\, e^{-\phi t}$$

It should be noted that the "rate at which work is being done" or the "effort" is monotonically decreasing and does not resemble either the Putnam or the Parr models. This behavior is not impossible for a small or trivial program (according to Fairley's classification scheme in Table II) whose initial problem complexity is negligible for a seasoned programmer.

Example II: Since Parr's main objection to the Putnam model was that the learning associated with the project was linear and continued to increase beyond the time in which the project was "considered complete", it should

be relatively easy to pick a more reasonable learning curve; one that reaches a finite limit. For example

$$9) \qquad q(t) = [\mu t/(1 + \beta t)]$$

is just such a function. We can normalize q(t) by making q(t) approach unity as time increases without bound. The solution of equation (1) with the learning function given above is

$$10) \qquad w(t) = 1 - X_0(1+\beta t)^{(\mu/\beta^2)} e^{(-\mu/\beta)t}$$

The behavior of this function is similar to the ones given by Putnam and Parr as can be seen in Figure II. It should be noted that by a suitable choice of parameters, the work done can start at zero or at some non-zero level as the modeler chooses. The rate of work done is given by

$$11) \qquad dw/dt = C\, e^{-\phi t} [(1 + \beta t)^{\Omega} - (1 + \beta t)^{\Omega-1}]$$

where $C = \mu X_0/\beta$ $\qquad \phi = \mu/\beta$ $\qquad$ and $\qquad \Omega = \mu/\beta^2$

The rate of work starts at C and has a shape similar to the Rayleigh distribution which is plotted in Figure II. A slight variation on this model would be to choose q(t) as given below;

$$12) \quad q(t) = e^{t}/(1 + e^{t})$$

This choice for q(t) would imply an easier project since the "learning" and hence the "available resources" increases faster than the one given above in Equation (9).

Example III: We can select a function for q(t) such as

13) $$q(t) = \beta(1 - e^{-\mu t})$$

The solution with the above choice is given by

14) $$w(t) = 1 - A \exp\{ - \beta t - (\beta/\mu)e^{-\mu t}\}$$

where $A = X_0 e^{\beta/\mu}$

The rate at which work is done is given by

15) $$dw/dt = A B \exp\{ - \beta(t + (1/\mu)e^{-\mu t}\}$$

where $B = \beta - \mu e^{-\mu t}$

We can normalize q(t) as before by choosing $\beta = 1$. In this case choosing $\mu=1$ will force dw/dt to start at zero. Typical behavior of this model is plotted in Figure III.

Summary: There are other intuitive interpretations that can be given to the models that are developed in this paper. As a starting point, we can differentiate equation (1) to obtain;

16) $$d^2w/dt^2 + \beta t\, dw/dt + \beta w = \beta$$

Putnam himself has noted that "this equation is similar to the nonhomogeneous second order differential equation frequently encountered in mechanical and electrical systems" {22}. The important difference is that the coefficient of dw/dt is not a constant but is an increasing function of time. The coefficient of dw/dt in physical systems is indicative of the

magnitude of the dissipation/friction term. If the model above is interpreted as "work=effort" equation then similar meanings can be attached to the various terms. Thus the dissipation/friction while work was being done would increase linearly in time, indicating that more and more effort is spent (one might say wasted) in trying to get the project completed. This effort might very well be the time spent in coordinating and various other activities that are not directly connected with coding--as can be seen from Table III. That the equation is being driven by 'learning constant' term in intuitively and logically pleasing. The fact that the dissipation term is a function of time can be attributed to the increasing difficulty of adding code the the project as time passes (and hence as it nears completion). Examined from this perspective, the models presented here result in more complex behavior that the Putnam model, as can be expected. The first model can be written as;

$$17) \quad d^2w/dt^2 + (\alpha+\beta t^{\mu})dw/dt + \beta\mu t^{\mu-1}w = \beta\mu t^{\mu-1}$$

The forcing term has now become a function of time along with the coefficient of w(t). The third example results in;

$$18) \quad d^2w/dt^2 + \beta(1- e^{-\mu t}) dw/dt + \beta\mu e^{-\mu t} w = \beta\mu e^{-\mu t}$$

In this example, the friction/dissipation term increases but not as rapidly as in the first example and it also has an upper limit of β, so that it approaches a constant instead of unconstrained growth as in the first example. Since in this last example, the "resources available" was modeled as $\beta(1-e^{-\mu t})$, it would also seem to imply that there is an upper limit to the resources and this clearly is the case in equation (18) where the forcing steadily decreases.

In summary, both the Putnam and Parr models can be represented as

$$19) \qquad dw/dt = q(t)[1 - w(t)]$$

where q(t) should reflect the characteristics of the project. If the project is a simple trivial program of the run-of-the-mill file processing type, q(t) might be a constant. If the project is a large, one-of-a-kind program, then q(t) might best be represented by w(t). This would imply that the programmers and analysts will be doing their learning along with the projects. Projects whose characteristics (i.e. difficulty) fall in between these two types of projects might be better modeled by choosing other functions for q(t) such as the ones presented above in this paper.

IV. STOCHASTIC MODELS

So far all of the models above have been deterministic. Furthermore, the models have focused on a single parameter w(t) which has been somwhat ambiguously defined as "essential completeness". It is for this reason that testing these models for correctness involves the time derivative which essentially translates into manpower. If however, we attempt to define "completeness" as the absence of bugs in the software, then the problem becomes rather difficult. Presently, several different approaches have been taken in attempts to solve this problem. There are static "Software Defects"

model which have been tested by Yu et al {29}, "Software Reliability" models (see for example Scholz {25}) and Markov Reliability model due to Siegrist{26}. It is very obvious to anyone in the field that no software package (especially a large one) can ever be completely free of bugs. Indeed, it is even difficult to have an estimate of the number of bugs. From Gilb {13} p. 33 we have;

> In real and large systems one would expect to find that there is a point of diminishing returns (in bugs found) on investment (of program debugging effort). Getting absolutely all the bugs out of a real program is now recognized as a task requiring an almost limitless amount of effort. This is not the place to explain why, but this assertion is supported by the fact that in 1972, in the 20th major version of their approximately 3 million statement "360 Operating System", IBM officially reported approximately 12,000 distinct new bugs in the system. At least 1,000 bugs had been discovered in each of the 20 "releases" in spite of 24-hour usage of the program for several years by thousands of computer installations.

With these considerations in mind, it would be possible to stochasticize the models due to Parr, Putnam and Hubey. In principle one would only need to add some random "shock" i.e. effect terms to the equations. Since every change to code can potentially result in some unintended (i.e. random) effect (especially if the older languages are used) on the rest of the software effect, a stochastic differential equation model will be a much better emulation of the software development effort. There are essentially two different approaches to incorporate random effects into a differential equation; "additive noise" and "multiplicative noise". The "additive" method perturbs the original equation by "adding" a random component usually "white noise". The multiplicative

From the standpoint of math. simplicity!
What is the justification in terms of softw. devel.?

method adds random effects to some or all of the coefficients of the equation. It would be best to solve for the most general case, however it is usually very difficult to get results. Only additive noise will be considered in this paper. The addition of a noise component will result in an equation of type

20) $$dw/dt = q(t)(1 - w) + \zeta(t)$$

where $\zeta(t)$ is Gaussian "white noise".

Why ?!?

No justification !?

It can be shown (see for example Jazwinski {15}, Soong {27} or Gardiner{12}) that the equation for the probability density p(x,t) of the stochastic process defined by the differential equation

21) $$dx/dt = f(x(t),t) + g(x(t),t)\ \zeta(t)$$

is given by the so-called Fokker-Planck Equation (usually called the *forward Kolmogorov equation* by mathematicians) given below

22) $$\partial p/\partial t = - \partial/\partial x \{f(x,t)p(x,t)\} + 1/2 \{\partial^2/\partial x^2 [g^2(x,t)p(x,t)]\}$$

Thus, the Fokker-Planck equation for Eq(20) is

23) $$\partial p/\partial t = - \partial/\partial w \{q(1-x)p\} + 1/2 \{\partial^2/\partial x^2\ p\}$$

Unfortunately, there are several problems with this formulation. First problem is that most equations of this type are generally difficult to solve explicitly. What is usually sought are some statistics about the solution process. Sometimes, the stationary probability distribution can be found. One can easily show that the equation for the stationary probability density p(w) is given by

$$24) \qquad \partial/\partial w\{\alpha\,(1-x)p_s(w)\} - 1/2\ \{\partial^2/\partial x^2\ p_s(w)\} = 0$$

Since the stationary probability density cannot be a function of time, we let lim $q(\infty)\to\alpha$ = constant and we set $\partial p/\partial t = 0$ in Eq.(23) to obtain Eq(24). It can be shown that the stationary density can be computed rather easily for such cases and is given by (See Soong [27]);

$$25) \qquad p_s(w) = C \exp\{ - 2\alpha w - \alpha w^2\}$$

where C is the normalization constant and can be evaluated to $C = e^{-\alpha}\sqrt{\alpha/\pi}$. To compute <w>, the first moment of w, it is easier in practice to obtain the characteristic function $P_s(k)$ which is the Fourier Transform of $p_s(w)$. The characteristic function--denoted by upper case P-- is given by

$$26) \qquad P_s(k) = e^{-\alpha} \exp\{\alpha\,[(2\alpha + ik^2)/(2\alpha)]^2\}$$

Now, the moment <w> can be computed from

$$27) \qquad \langle w\rangle = (1/i)\,\partial/\partial k\,(P_s(k)_{k=0} \dashrightarrow 1$$

The first moment <w> has been shown to be unity. This result was to be expected since the added noise is symmetrical (i.e. Gaussian) and the

original equation is linear. Since the solution is symmetric this implies that there will be finite probability of greater than 100% completion which is not meaningful. What should really be done is that a one-sided noise component should be added to the software development equation. If non-white noise is used, the equation for p(w,t) as in Eq.(23) is no longer valid since the process is not Markovian. To remedy this situation one needs to construct a two-dimensional Markov process one of whose components is the software development process equation. An acceptable process to generate nonwhite noise is given by

Justify in terms of "bugs", rather than math !!

28) $$dy/dt = -\gamma + \zeta(t)$$

where $\zeta(t)$ as before is a Gaussian white noise. It can be shown as done above that the stationary distribution for y is exponential in y

29) $$p_s(y) = C\, e^{-2\gamma y}$$

If we now append to Eq.(28), the software development equation, driven by the noise y(t) as given by

30) $$dw/dt = \alpha\,(1-w) + \xi\, y$$

we will have a two-dimensional vector Markov process whose stationary joint probability density is given by

31) $$-\partial/\partial x_1\{\alpha\,(1-x_1)+\xi x_2\}p_s(x_1,x_2) + \partial/\partial x_2\{-\gamma\, p_s(x_1,x_2)\} + D\partial^2/\partial x_2{}^2\{p_s(x_1,x_2)\}=0$$

Simplifying this equation and taking the Fourier Transform with the kernel exp $\{i(k_1x_1 + k_2x_2)\}$ results in a first order ordinary differential equation for the joint characteristic function

32) $$dP_s/dk_1 - \{(i(\alpha + \gamma)k_1 - D^2 k_2^2)/((\alpha + \gamma + \xi)k_1)\}P_s = 0$$

Upon integrating we obtain

33) $$P_S(k_1,k_2) = \Omega\, k_1^A\, e^{Bk_1}$$

where $$B = (i\alpha)/(\alpha+\gamma+\xi)$$

$$A = - D^2k_2^2)/(\alpha+\gamma+\xi) - ik_2(\alpha+\gamma)/(\alpha+\gamma+\xi)$$

Ω is a constant of integration and i is the square root of unity. This characteristic function $P_S(k_1,k_2)$ can be inverted to obtain the probability density of the process. However it is much easier to compute the moments from the characteristic function as before. In particular, the first moment is given by

34) $$\langle x1 \rangle = \langle w \rangle = (\Omega / i) \{ e^{Bk_1} \partial/\partial k_1 \; k_1^A + B\, k_1^A\, e^{Bk_1} \}_{k=0}$$

With Ω =1 and $k_2 \to 0$ and $k_1 \to 0$, it can be shown that the moment is given by

35) $$\langle w \rangle = ((\alpha+\gamma)/(\alpha+\gamma+\xi) < 1$$

Since γ and ξ have to do with the noise magnitude, we would like to keep them as small as possible. Hence in order for the project to be 100%

strong; but heavily dependent on assumptions not fully justified!

complete we must have $\alpha \to \infty$ which implies that the complete removal of all bugs from a software package cannot be done without essentially an infinite amount of resources--(in time and space).

V. CHAOS IN SOFTWARE PRODUCTION

So far all of the models considered have been continuous in time. In reality, however, a discrete model would be much more appropriate since programmers' effort probably is best measured in terms of days, weeks or hours. A much more interesting case arises when one attempts discrete formulation. One can model the progress on the project by a discrete model such as W_n .

It is not too difficult to come up with a models as below;

36a) $$W_{n+1} = W_n + aW_n - bW_n^2 = a W_n - b W_n^2$$

36b) $$W_{n+1} = kW_n(1 - W_n)$$

It should be noted that Eq. (36b) is a discretized version of the Parr model. In both models above one assumes that the amount of work done (completed) W_{n+1} is derived from the previous period by adding a term proportional to the work done during last period (i.e. period n) and adding a dissipation term proportional to the square of the W_n term. The coefficient k in Eq (36b) can be thought of as a "time-compression" term as used by Parr and Putnam to address the difficulty of the project. The equation above (36b) is quite notorious for its strange and totally unexpected behaviour (see

for example May {18}, Devaney {9} and Jensen {16}). It is called the logistic equation or sometimes called the Feigenbaum Oscillator after the person who elucidated many of its properties. There has been a whole book devoted to it (Collet and Eckmann {10}). Hence it will be impossible to do justice to this model in a short paper. However, some salient features will be noted. For small values of k (not too small), $W_\infty \to 1$, so that it can be thought of as representing a run-of-the-mill software project. For increasing values of k, it converges more quickly than before so that we can think of the parameter k as a kind of a "pressure" or "hurry" parameter. At certain values of k, however things began to change. There is no longer a convergence but a continuous oscillation between values determined by the size of the coefficient k. As the value of the coefficient k is increased, the behaviour of W gets strange in the sense that it goes through what is called a "period doubling" bifurcation; that is for a given value of k, W oscillates between two values, then when k is increased W oscillates between four values, etc. Finally at a value of k=4, there is no periodicity left at all but rather a seemingly random or "chaotic" oscillation. This "incapability" to converge to unity is a very useful property for modeling the software production process. The spontaneous production of utter chaos from completely deterministic iteration is a lot like the production of "unfinishable" software packages no matter how much resources are poured into it. Since during the software production process there is really not randomness at all, and since the software modules are completely prescribed by deterministic rules which get executed on deterministic

Do you mean soft. production is deterministic ?!?

machines, it is not difficult to see why the logistic model is a tantalizing one for modeling the Software Production Process. Although it is difficult to justify the period doubling bifurcation to chaos, in reference to Software Production, there are really no other models which are not ad hoc in the way in which the randomness enters the model. The logistic models are the only ones which have this property. Perhaps "Chaos" is not an inappropriate word for the most difficult "embedded" software systems at all.

VI. AFTERWORD

Several deterministic and stochastic one parameter models resembling those of Putnam and Parr have been presented. It has been shown that the deterministic models converge to unity as $t \to \infty$. Probabilistic aspects of the software production process involve making unintended (hence random) changes to the software package. Models that take into account the random errors creeping into the software production process have been developed using methodology due to Fokker-Planck and Kolmogorov. It has been shown with these stochastic models that---in agreement with practice and intuition--- a software package can only be completely bug-free with only infinite amount of resources and an infinite amount of time. The implication of this, of course, should be to develop methodology that takes the randomness out of the software production process. In another vein, a discrete model has been developed which has some of the characteristics of the logistic growth models. This discrete model has disturbing implications that even if there is no randomness, it is still possible not to be able to finish and to wind up

Explain better! What do you mean?

So, why do they deserve attention?

with an unusable and unreliable system. None of these models have yet been tested on real software projects. Hence, some field data needs to be in before any predictions can be made as to their validity. The ramifications of the stochastic and chaotic models in this paper will have to be treated more fully in the future.

REFERENCES

1. Bairdain, E., "Research Studies of Programmers and Programming", unpublished study, New York, 1964

2. Boehm, B., Software Engineering Economics, Prentice-Hall, Englewood Cliffs, New Jersey, 1981.

3. Braun, M., Differential Equations and Their Applications: Applied Mathematical Sciences, Vol 15, Springer-Verlag, New York, 1983

4. Brooks, F., The Mythical Man-Month, Addison-Wesley, Reading, Massachusetts, 1975

5. Buffa, E., Modern Production Management, Wiley, New York, 1969

6. Comer, D., Operating Systems I: The XINU Approach, Prentice -Hall, Englewood Cliffs, New Jersey, 1987

7. Comer, D. and M. Halstead, "A simple experiment in top-down design", IEEE Trans. on Software Engineering, Vol SE-5, pp. 105-109, May 1979

8. Davis, H., Introduction to Nonlinear Differential and Integral Equations, Dover Publications, New York, 1962

9. Devaney, "Chaotic Bursts in Nonlinear Dynamical Systems", Science,

16 January 1987, pp. 342-345.

10. Eckmann, J.P and P. Collett, Iterated Maps on the Interval as Dynamical Systems, Birkhouser, Boston, 1980.

11. Fairley, R., Software Engineering Concepts, McGraw-Hill Publications, New York, 1985.

12.Gardiner, Handbook of Stochastic Methods, Springer-Verlag, Berlin, 1983.

13. Gilb, T., Software Metrics, Winthrop Publishers, Cambridge, Massachusetts, 1977.

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

15. Jazwinski, A., Stochastic Processes and Filtering Theory, Academic Press, New York, 1970.

16. Jensen, R., "Classical Chaos", American Scientist, Volume 75, March-April 1987, pp.168-181.

17. Martin, J., Fourth Generation Languages: Volume I, Principles, Prentice-Hall, Englewood Cliffs, New Jersey, 1985

18. May, R., "Biological Populations with Nonoverlapping Generations:Stable Points, Stable Cycles and Chaos", Science, Vol. 186, 15 November 1974, pp. 645-647

19. Mohanty, S., "Software Cost Estimation: Present and Future", Software-Practice and Experience, Vol 11, pp.103-121, 1981

20. Parr, F.N., "An Alternative to the Rayleigh Curve Model for Software Development Effort", IEEE Trans. on Software Engineering, Vol. SE-6, No.3,

May 1980.

21. Perlis, A., F. Sayward and M. Shaw, Ed, Basili, V., "Resource Models", .in Software Metrics, , The MIT Press, Cambridge, 1981.

22. Putnam, L., "A General Empirical Solution to the Macro Software Sizing and Estimating Problem", IEEE Trans. on Software Engineering, Vol. SE-4, No. 4, July 1978.

23. Rudolph, E., "Productivity in Computer Application Development", Auckland, New Zealand: University of Auckland, 1984.

24. Sackman, H., et al., "Exploratory Experimental Studies Comparing Online and Offline Programming Performance", Communications of the ACM, Vol 11, No. 1, January 1968.

25. Scholz, F., "Software Reliability Modeling and Analysis", IEEE Trans. on Software Engineering, Vol SE-12, No.1, January 1986, pp. 25-31

26. Siegrist, K., "Reliability of Systems with Markov Transfer of Control", IEEE Trans. on Software Engineering, Vol SE-14, No.10, October 1988, pp. 1049- 1053.

27. Soong, T.T., Random Differential Equations in Science and Engineering, Academic Press, New York, 1973

28. Tanenbaum, A., Operating System Design and Implementation, Prentice-Hall, Englewood Cliffs, New Jersey, 1975.

29. Yu, T., Vincent Shen and H. Dunsmore, "An Analysis of Several Software Defect Models", IEEE Trans.on Software Engineering, vol SE-14, No. 9, September 1988, pp.1261-1270.

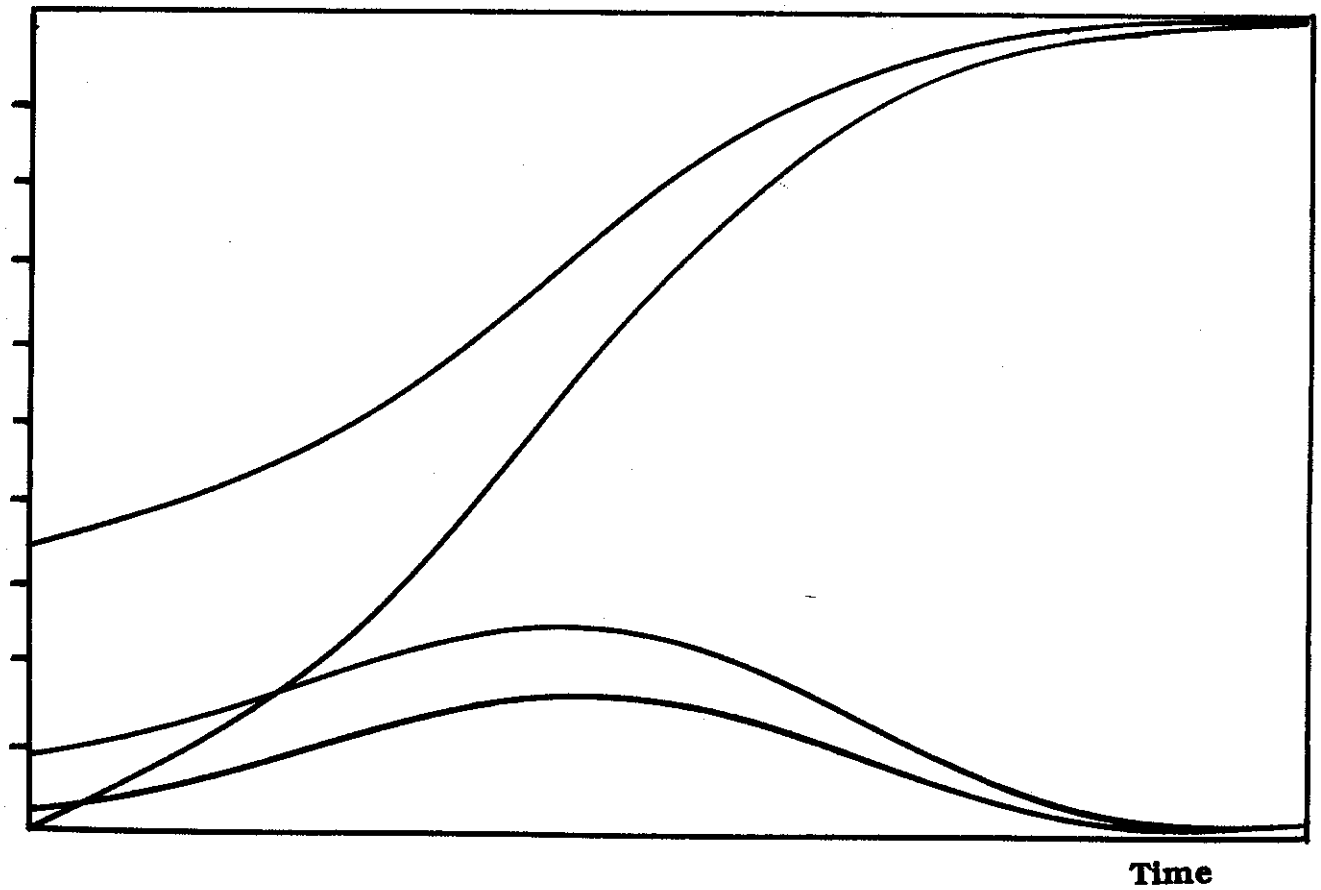

FIGURE I

$q(t) = \alpha + \beta t^{\mu}$

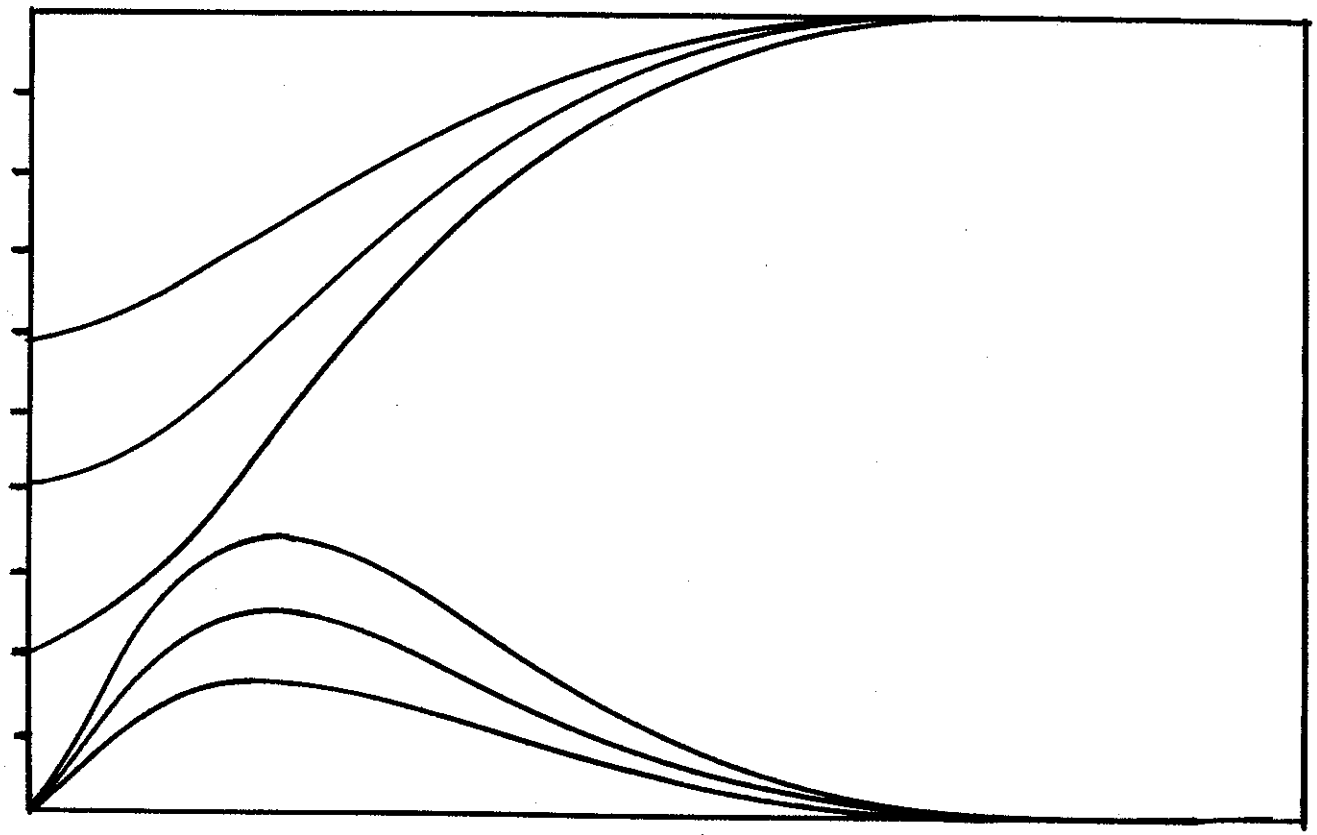

FIGURE II
$q(t) = \mu t/(1 + \beta t)$

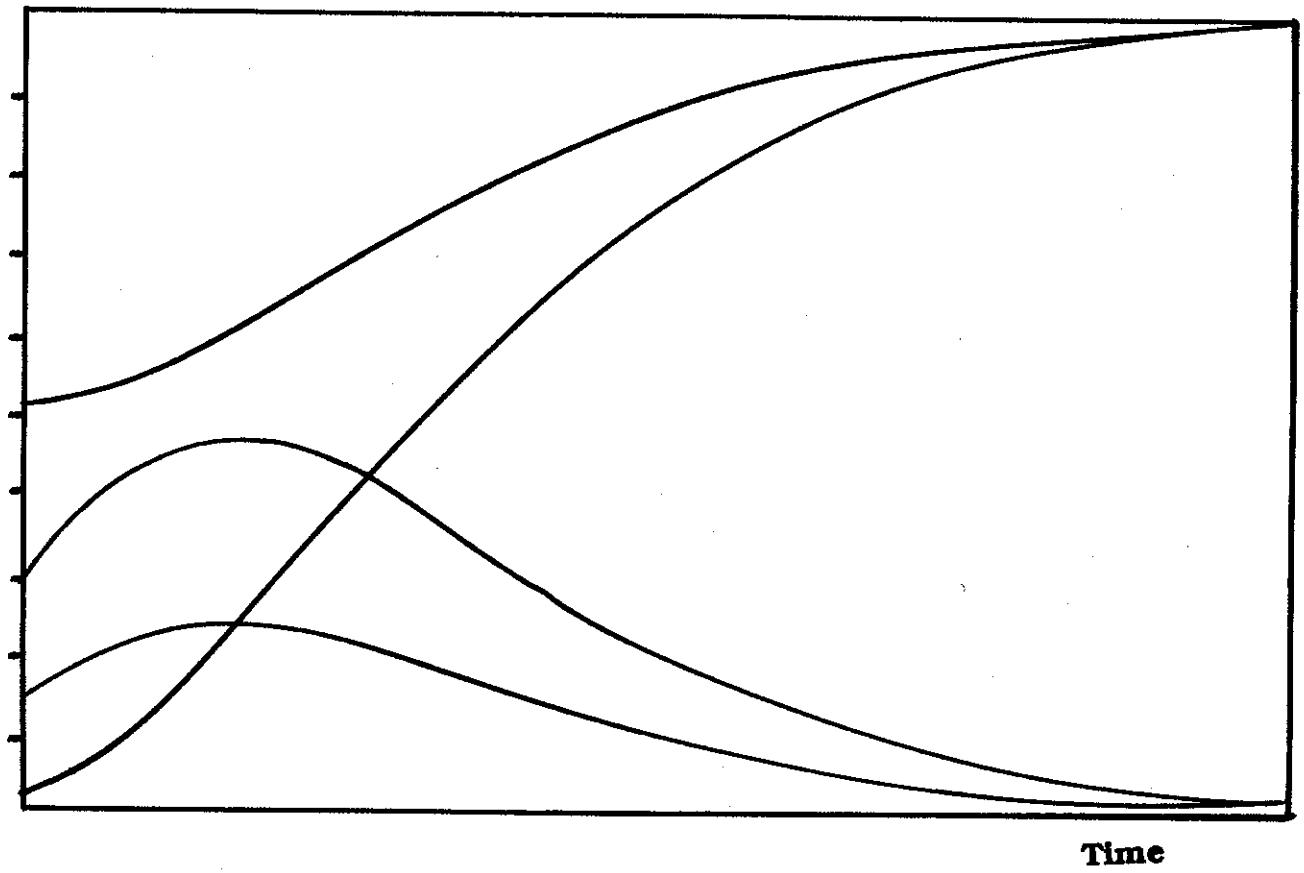

FIGURE III

$q(t) = \beta(1 - e^{-\mu t})$

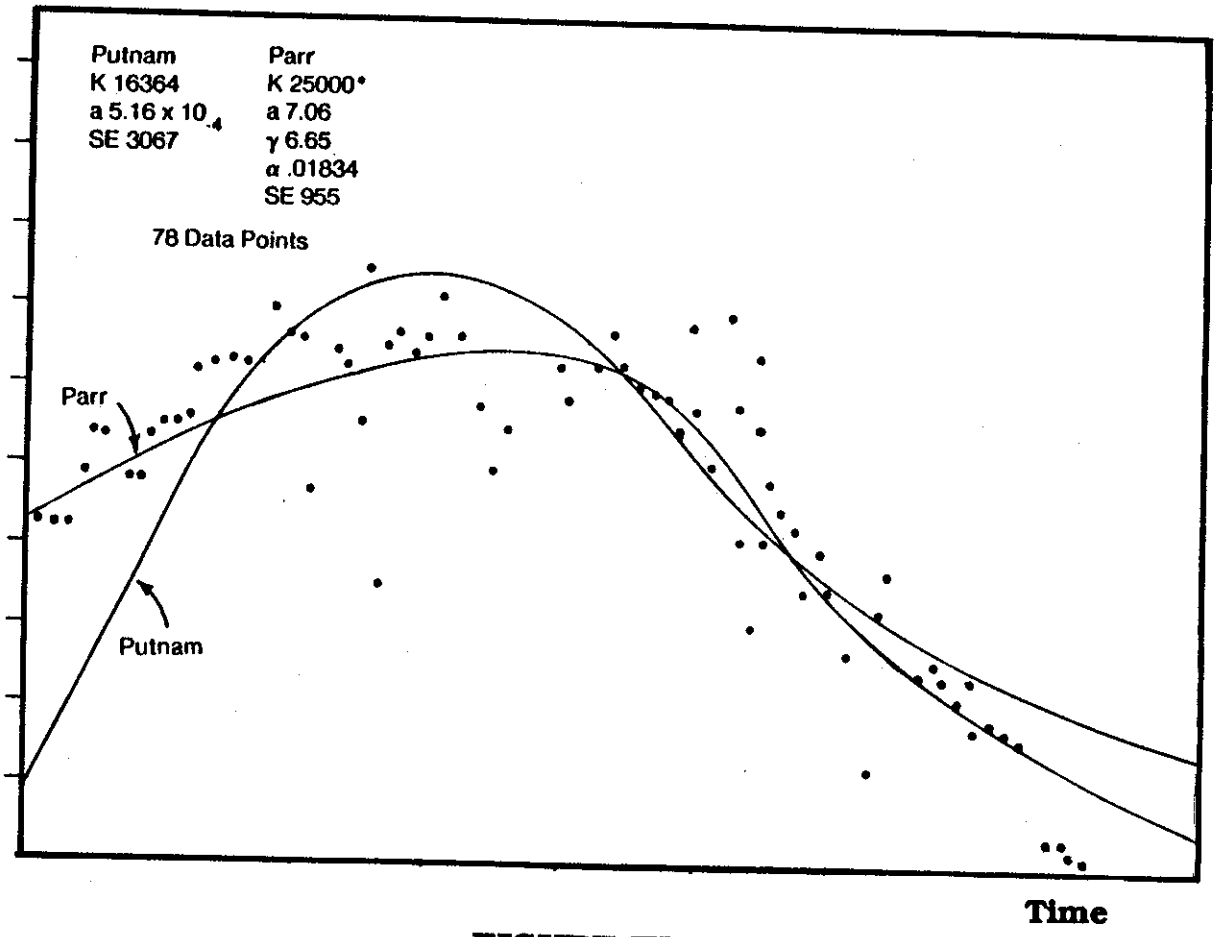

FIGURE IV

From V. Basili in Perlis, Sayward & Shaw in Software Metrics