JOURNAL OF AEROSPACE COMPUTING, INFORMATION, AND COMMUNICATION
Vol. 5, March 2008

Software Production Process for Safety Critical Software

Norman Schneidewind*
Professor Emeritus, Naval Postgraduate School, Monterey, CA, 93942

DOI: 10.2514/1.34021

A software production model is developed that integrates process and product and is designed to identify bottlenecks in the production process. Choke points can occur either from process deficiencies or failure to identify and correct extant software defects. NASA Goddard Space Flight Center satellite defect data is used in the analysis because the primary aim is to apply the model to safety critical software. The model applies feedback control to correct anomalies in process and product that may occur. Both predictions and actual defect data are used to identify process and product behavior that do not meet expectations.

I. Introduction

ACCORDING to,[1] software systems come and go through a series of passages that account for their inception, initial development, productive operation, upkeep, and retirement from one generation to another. We focus on the development and production phases and introduce something different in this research compared with articles about software development process that correctly relate the effectiveness of the process to the quality of the product. We go a step further and introduce the concept of the *reliability* of the process, both current and *future*. We perform this analysis by using software defect data from the NASA Goddard Space Flight Center satellite project known as JM1. These data are shown in Table 1. Our goal, recognizing the importance of integrating process and product,[2] is to identify production bottlenecks in the process that cause product delivery to be delayed. In addition, we identify deficiencies in the defect removal process that can lead to product unreliability and delivery schedule delay.

The principal features of the production system are defined using process flow charts illustrated in Fig. 1 for a software production and quality control operation. The flow of software modules is from left to right. Each phase has a processing time measured in days and a defect rate. Our model assumes that defects introduced by a process step *may* pass through downstream steps undetected until an inspection is performed.[3]

A. Comparison of Hardware and Software Production Processes

Some authors claim that software engineering is significantly different from software engineering. For example, in,[4] the authors claim that in hardware, the concern is with control of manufacturing defects; assessment of mean time to failure of a product through wear or aging; and use of statistical sampling to provide quality predictions in an environment of defined uncertainty. In general, these have limited applicability in software engineering. The main reason for this is that in software engineering we are concerned with controlling the design process and not the manufacturing process. We disagree because hardware engineering is not confined to controlling for manufacturing defects. It is also concerned with controlling for design defects (e.g., semiconductor products), as in the case of software.[3,5,6] Secondly, software can wear and age, for example, the classic case of buffer overflow and the well-known problem of having to restart a PC in order to reestablish a stable state. Thirdly, we suggest that noise, temperature, electrical system irregularities, and the harshness of space, do not constitute defined uncertainties.

Received 13 August 2007; accepted for publication 19 February 2008.
*ieeelife@yahoo.com

Table 1 NASA Goddard Space Flight Center JM1 Satellite Defect Data.

phase	I	t_i	d_i	D_i	j	t_j	d_j	D_j
requirements analysis	1	6.6156	78	0.1902	1	65	26	0.4000
design	2	11.1708	8	0.0195	2	57	20	0.3509
coding	3	0.9685	20	0.0488	3	187	14	0.0749
development test	4	0.6577	161	0.3927	4	341	10	0.0293
integration	5	24.1259	4	0.0098	5	331	15	0.0453
reliability analysis	6	2.0246	33	0.0805	6	351	11	0.0313
system test	7	37.786	3	0.0073	7	66	14	0.2121
field deployment	8	6.6914	98	0.2390	8	62	10	0.1613
post-release	9	8.4667	5	0.0122	9	178	13	0.0730
					10	61	7	0.1148
					11	98	10	0.1020
					12	56	9	0.1607
					13	35	10	0.2857
					14	34	12	0.3529
					15	18	8	0.4444
					16	41	7	0.1707
					17	41	9	0.2195
					18	398	3	0.0075
					19	380	4	0.0105
					20	377	6	0.0159
					21	373	8	0.0214
					22	367	8	0.0218
					23	365	6	0.0164
					24	365	9	0.0247
					25	359	5	0.0139
					26	357	5	0.0140
					27	356	7	0.0197
					28	501	6	0.0120
					29	355	6	0.0169
					30	355	5	0.0141

Actually, with modification to account for software's peculiarities (e.g., large number of execution paths), we find the hardware model production process model to be a useful guide for conceptualizing the software production process in Fig. 1.

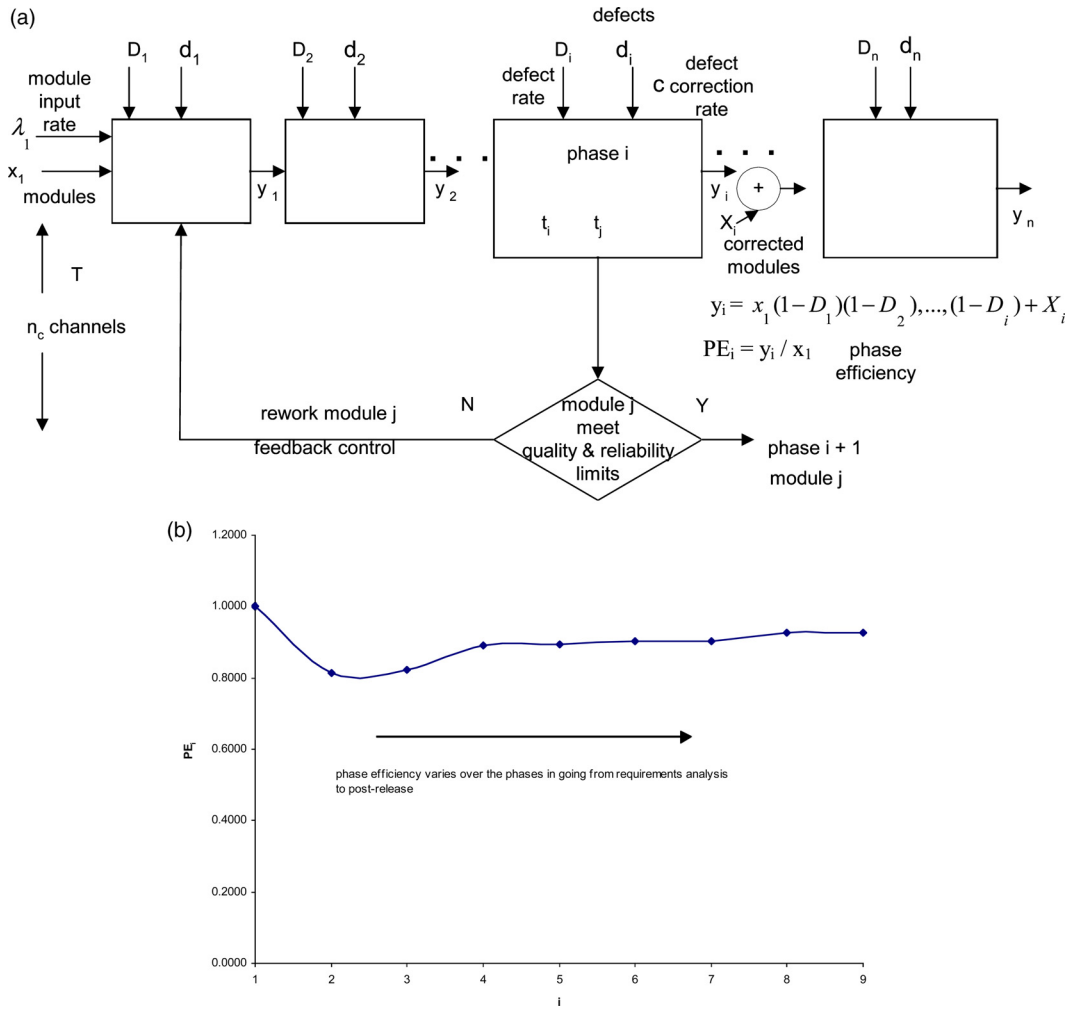
In the following subsections we describe the important characteristics of the software production process, based on other researchers' work that we address in our research.

1. Process Flow Analysis

The processes and activities associated with production are used in analyzing and improving process flows of software and information about software. Flow analysis is an abstraction of the production process that neglects the detail concerned with individual items of product, but considers the collection as a flow. Thus most of the parameters of the model relate to averages over time. One of the primary purposes of a flow-process chart like the one in Fig. 1 is to identify bottlenecks and to exert process and product quality control.[5,6]

2. Product Flow Analysis

A product comprises all artifacts that describe the software being produced. These include the requirements, designs, code, etc. to be delivered to the customer. Products have the following attributes: content, which represents some measure of 'how much is there'; quality, which represents 'fitness for purpose'; and cost, which represents the various costs (e.g., time consumed in performing product activities). As the project progresses, changes occur according to the phases carried out in the process.[7] In Fig. 1, product attributes are defects, defect rate, and time spent in phases and in correcting defects.



$$y_i = x_1(1-D_1)(1-D_2),...,(1-D_i)+X_i$$

$$PE_i = y_i / x_1$$

Fig. 1 Software production process; NASA JM1 software system efficiency phase efficiency vs. software production phase.

3. Feedback Control

Feedback control of the software production process is shown in Fig. 1 wherein control is exercised by reworking modules that do not meet quality and reliability limits. This concept is in the spirit of Miller et al.[8] As a discipline, control engineering develops strategies, which are capable of driving the behavior of a software system to a desired target behavior, such as meeting reliability specifications. Feedback control refers to the set of techniques that use knowledge of the software system states (e.g., state of module reliability) and outputs (e.g., corrected modules) to determine a set of future module and defect inputs that will lead to the desired target behavior.

Cangussu et al. model consists of two components: a feedback control portion and a model parameter estimation portion.[9] Their focus was on the assessment of the goodness of the parameter estimates and model predictions and their utility in the management of the system test phase. Estimates of the number of residual defects are used to control the quality of the product being released. Estimates of the number of defects in the application when the test phase began and at the current checkpoint were obtained. In addition, predictions were made of the reduction in the number of remaining defects. The parameter estimates and reliability predictions assist management in planning the test. Our research relates the attributes of software reliability model parameter estimates to defect reduction prediction accuracy. When sufficient accuracy is obtained, the test phase exerts feedback to control module quality in Fig. 1.

4. Characteristics of the Test Phase

Over the course of the test phase, a manager establishes checkpoints where he or she can observe quality measures and compare these with those expected by specifications. According to this comparison, the manager decides for changes in the test environment necessary to better meet reliability objectives. A software production model will provide quantitative information to the manager about the evolution of the test process relative to the specification and can identify specific changes necessary to meet a pre-specified quality objective.[10] We will make comparisons of both specified process and product reliability with achieved reliability when our model results are analyzed.

II. Software Production Process Model Development

First some definitions that are used in the equations that follow and in Fig. 1.

A. Definitions

1. Numbers

x: number of modules inputted to phase 1 = demand for module production at each phase
rm_i: remaining number of modules in phase i that has not been processed due to the presence of defects
y_i: number of modules output of facility i
X_i: expected number of modules corrected in phase i
d_i: number of defects *discovered* in phase i
d_j: number of defects per module j
dc_j: number of defects that have been *corrected* for module j
rd_j: remaining number of defects that has not been corrected for module j
N: total number of modules in software release
n: number of modules in sample ($n < N$)
N_c: number of modules corrected in software release
N_p: number of phases
n_c: number of parallel production channels

2. Rates

λ_1 : module mean input rate to phase 1 (modules per day)
D_i: defect *discovery* rate in phase i (defects per day)
D_j: defect *discovery* rate in module j (defects per day)
c: defect correction rate (defects per module)

3. Times

t_i: expected time a module spends in phase i (days)
t_j: time required to correct defects in module j (days)
T: expected time spent in system correcting defects (days)
t: Future time when predicted quantity is to occur

4. Probabilities

p (t_i): probability of spending t_i time in phase i
p (c): probability of correcting defects

5. Metrics

PE_i: phase efficiency in phase i
R_i: process reliability of phase i
R_j: product reliability of module j
R_g: reliability goal (e.g., .95)
R (t): predicted reliability to occur at time t
F (t): *process* cumulative defects predicted to occur at time t_i

6. Software Reliability Model[11]

α: Failure rate at the beginning of interval s
β: Negative of derivative of failure rate divided by failure rate (i.e., relative failure rate)
s: Starting interval for using observed failure data in parameter estimation
X_{s-1}: observed failure count in the range $[1, s-1]$
T: total time spent correcting defects in modules in n_c channels
t_i: expected time a module spends in phase i
t_j: time required to correct defects in module j

B. Equations

Equations are classified by those used in production and defect analyses for product and process and those used in software reliability prediction.

1. Production and Defect Analysis for Process and Product

The defect discovery rate in phase i is given by:

$$D_i = \frac{d_i}{\sum_{i=1}^{N_p} d_i} \tag{1}$$

And the defect correction rate is computed by:

$$c = N_c/N \tag{2}$$

The defect discovery rate in module j is:

$$D_j = d_j/t_j \tag{3}$$

It is important to exercise control over the quality of modules produced in the production process as shown in Fig. 1. Therefore, from equation (3), we compute the upper (UCL) quality control limit from the mean and standard deviation:

$$UCL = \bar{D}_j + 3SD(D_j) \tag{4}$$

We want to estimate the expected number of module inputs corrected in phase i, based on the number x_1 that are inputted to phase 1, and the probability of correction, as follows:

$$X_i = x_1 p(c) \tag{5}$$

Now we assume that the time to correct defects in phase i t_i is exponentially distributed, so that the probability of defect correction is:

$$p_i(c) = D_i e^{-D_i t_i c} \tag{6}$$

Then using equations (5) and (6), the expected number of modules corrected in phase i is equal to:

$$X_i = x_1 D_i e^{-D_i t_s c} \tag{7}$$

Then using equation (7), the expected output in phase i, based on the loss in output due to defect and the gain attributed to defect correction, is computed by:

$$y_i = x_1(1 - D_1)(1 - D_2), \ldots, (1 - D_i) + X_i \tag{8}$$

Then the remaining module input in phase i that has not be processed due to the presence of defects is:

$$r_i = x - y_i \tag{9}$$

In Fig. 1, the output at the facility i should obey the demand constraint (i.e., input at phase1) this phase:

$$y_i \geq x_1 \tag{10}$$

We need to compute the input rate of modules N into the system in phase 1, using the total time to correct defects T:

$$\lambda_1 = \mathrm{N/T} \tag{11}$$

Phase efficiency in phase i is determined by the relationship between output and demand:

$$\mathrm{PE_i} = y_\mathrm{i}/x_1 \tag{12}$$

Assuming the time spent in phase i is exponentially distributed, and using equation (11), we have the probability of this time:

$$\mathrm{p(t_i)} = \lambda_1 e^{-\lambda_1 t_i} \tag{13}$$

The expected time modules spend in the system *in all channels* correcting defects, as shown in Fig. 1, is estimated by:

$$\mathrm{T} = \sum_{j=1}^{m} t_j \tag{14}$$

Since we do not want modules to be languishing in the production system for more than, let us say 365 days, the number of channels required (rounded up) is computed by:

$$\mathrm{n_c} = \mathrm{T}/365 \tag{15}$$

Using equation (6) we estimate the expected number of defects corrected in phase i as:

$$\mathrm{dc_i} = \mathrm{d_i p(c)} = \mathrm{d_i}\ D_i e^{-D_i t_i c} \tag{16}$$

Next, using equation (16), we compute the remaining defects, that is, the number that have not been corrected in phase i:

$$\mathrm{rd_i} = \mathrm{d_i} - \mathrm{dc_i} \tag{17}$$

It is important to assess the empirical reliability from two standpoints: 1) with respect to the *process* in each phase i and 2) with respect to the *product* for each module j.

For 1) we have:

$$\mathrm{R_i} = 1 - \left(\frac{d_i}{\sum_{i=1}^{n} d_i}\right) \tag{18}$$

And for 2) we have:

$$\mathrm{R_j} = 1 - \left(\frac{d_j}{\sum_{j=1}^{m} d_j}\right) \tag{19}$$

2. Software Reliability Prediction

We use two approaches to making predictions about *future* process and product reliabilities because experience has shown that a single approach does not always achieve minimum prediction error. One approach uses the minimization of the likelihood function to estimate model parameters.[12] The second approach uses the SMERFS software reliability tool to operate on the likelihood function using a minimum least square errors criterion.[13] Both approaches were used for all predictions. In all cases, except for the product reliability prediction, the first approach was superior. For the one exception, we provide both predictions.

Using the first approach, we estimate the model parameters α, β, and s, from the process and product defect data, and predict reliability in equation (20) with a C++ program we wrote.[14] Both process and product reliability are

predicted, where the time t will be t_i for the process case and t_j for the product case in equation (20). In the second approach, the SMERFS tool is employed, using the same procedure.

$$R(t) = e^{-[\frac{\alpha}{\beta}[e^{-\beta(t-s+1)}-e^{-\beta(t-s+2)}]]} \tag{20}$$

A second prediction metric is cumulative defects shown in equation (21). We made predictions using the two approaches for process defects. We were unable to obtain realistic predictions (i.e., excessive prediction error) for product cumulative defects.

$$F(t) = (\alpha/\beta)\left[1 - e^{-\beta(t-s+1)}\right] + X_{s-1} \tag{21}$$

C. Results of Software Production Analysis

We present a series of plots designed to illuminate important characteristics of process and product efficiency and reliability. First, in Fig. 1, we show phase efficiency plotted against phase that indicates that the efficiency of the design and coding phases is relatively low. This means that output is low relative to input for these phases. This could be caused either by low productivity in theses phases or by a requirement to design and code complex software.

Next, in Fig. 2, we want to determine whether the probability of correcting defects tracks the probability of time spent in phases, across phases. In Fig. 2 this is the case. If this had not been the case, it would be indicative of problems in the defect correction process.

Now the focus shifts to the correction of defects in modules, as shown in Fig. 3. Here we provide an upper control limit (UCL) computed from the mean and one standard deviation of the defect rate. The plot delineates which modules meet the quality standard and which do not. In addition, the figure indicates that it would be infeasible to meet the module processing requirements in a single production channel. Thus, in Fig. 3 we have computed the number of parallel channels needed. These channels are depicted in Fig. 1.

Figure 4 reveals an alarming situation: for the development test phase, defect correction is way behind defect discovery. This could be the result of ineffective testing in this phase or modules that reach this phase that are defect prone.

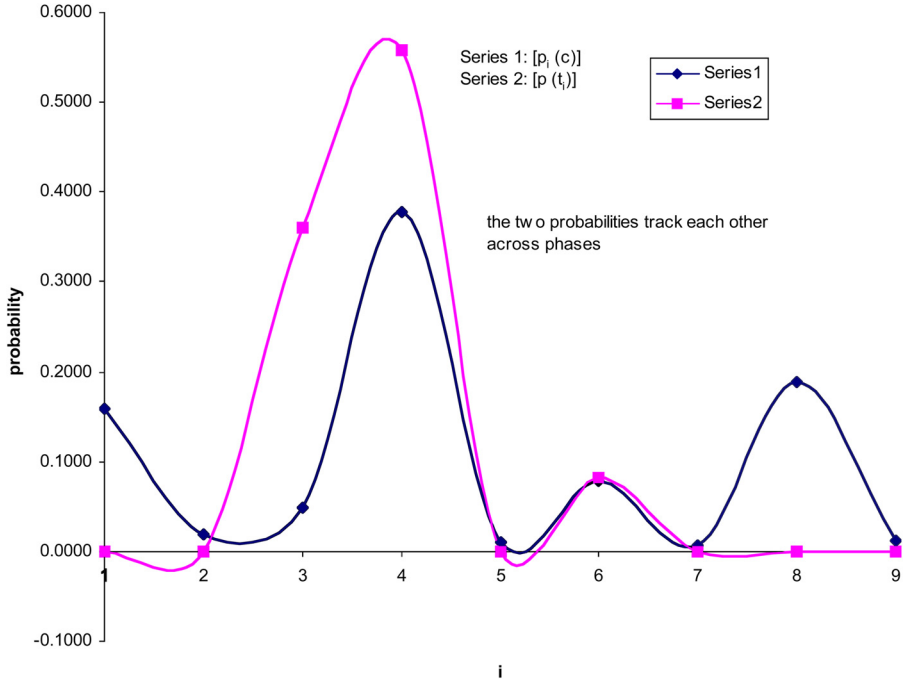


Fig. 2 NASA JM1 software: probability of spending time t_i in phase i[$p(t_i)$] and probability of correcting defects in phase l[$p_i(c)$] vs. i.

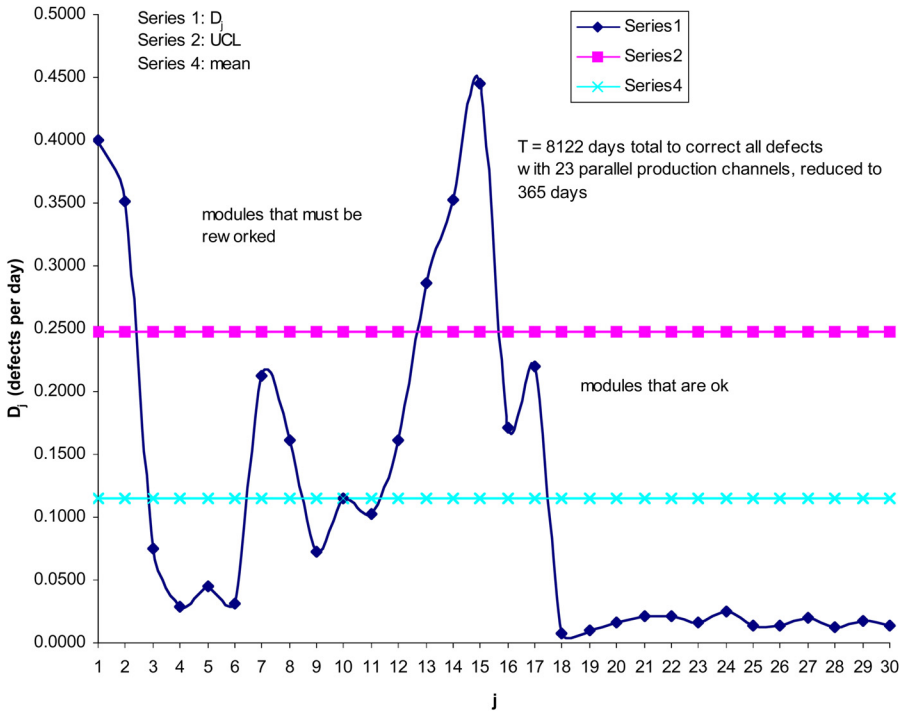


Fig. 3 NASA software JM1: module defect rate D_j vs. module number j.

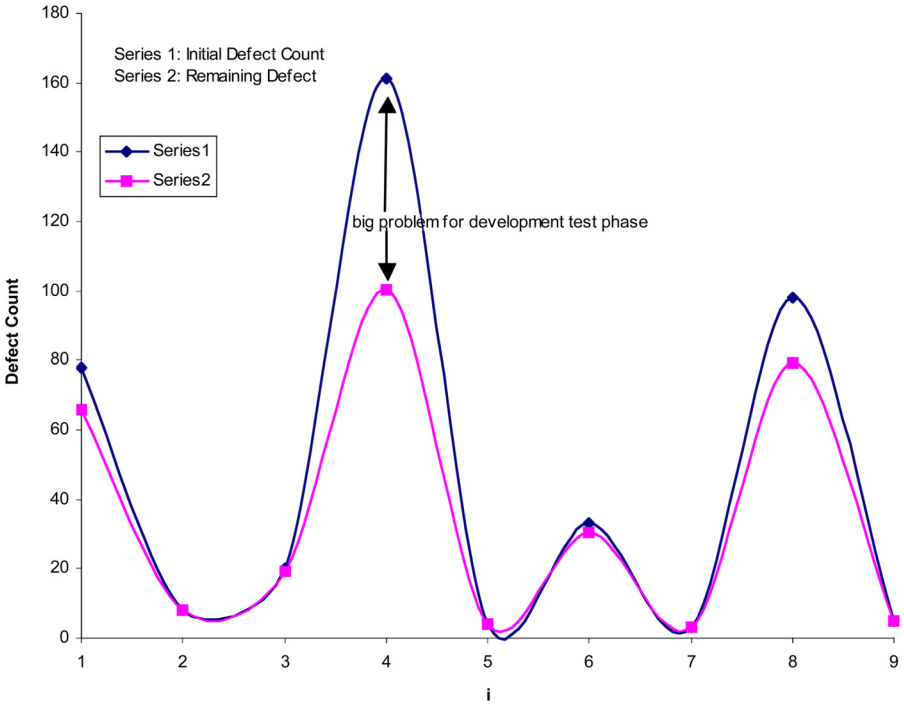


Fig. 4 NASA software JM1: defect count vs. phase i.

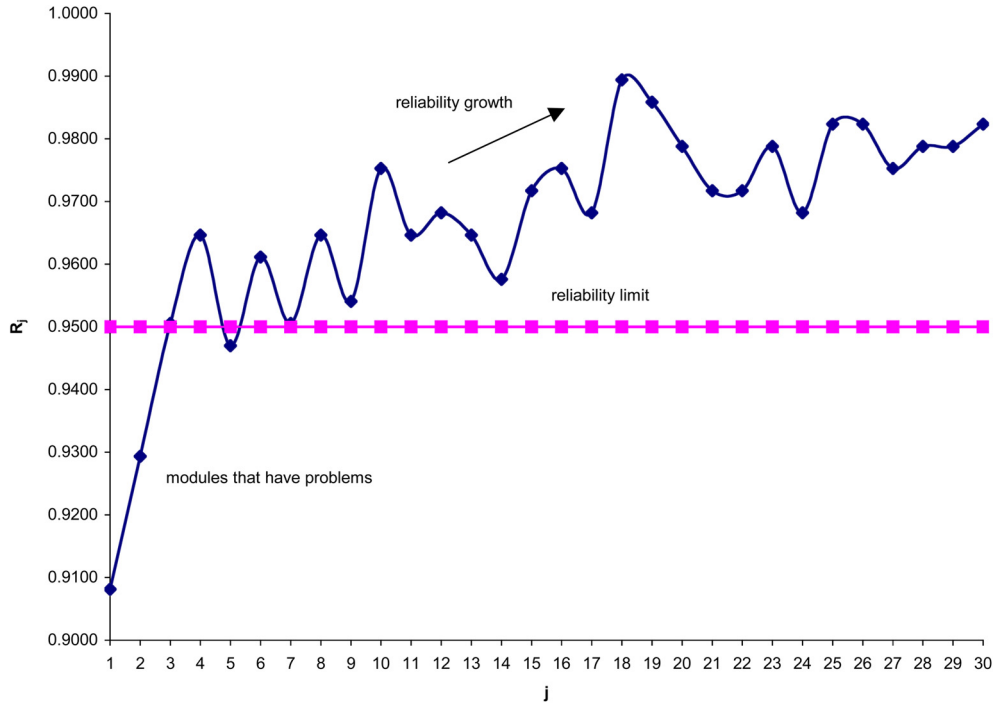


Fig. 5 NASA JM1 software: actual product reliability R_j vs. module j.

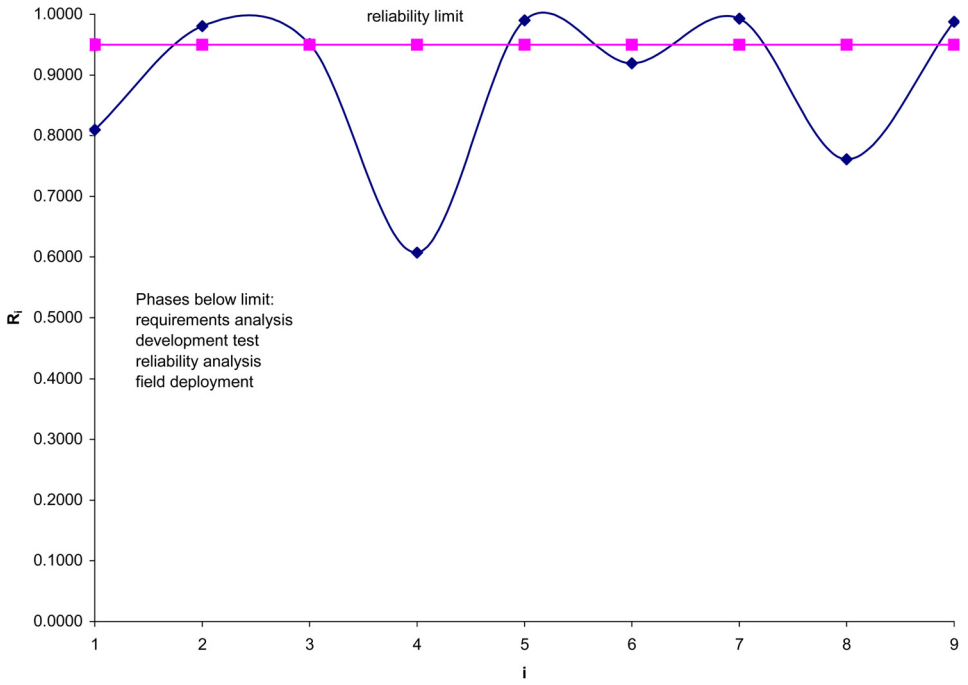


Fig. 6 NASA JM1 software: process reliability R_i vs. phase i.

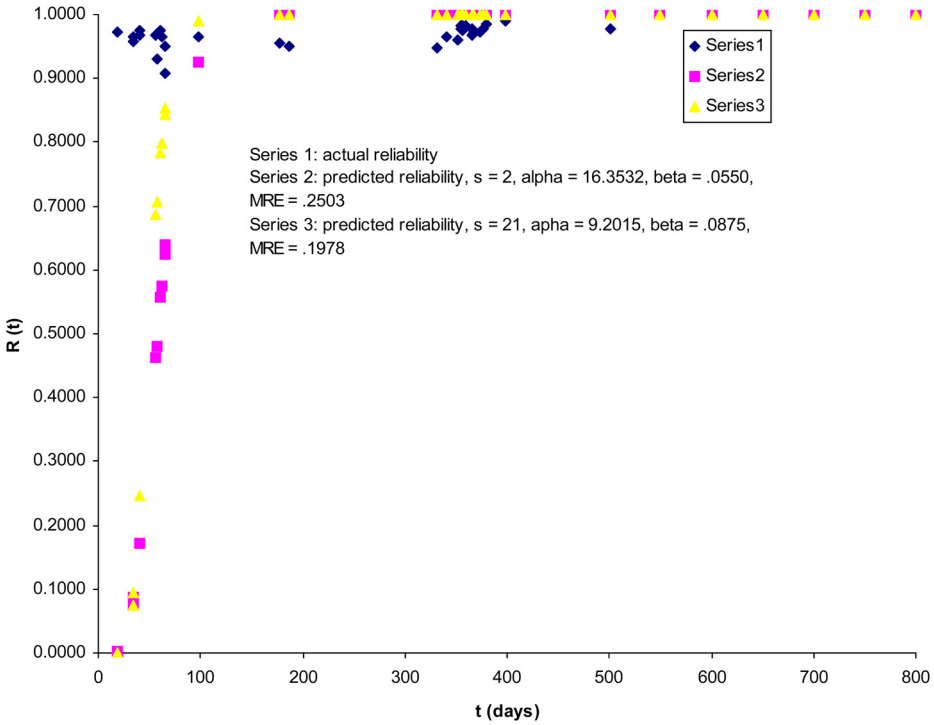


Fig. 7 NASA JM1 software: product reliability for module j R(t) vs. time *t*.

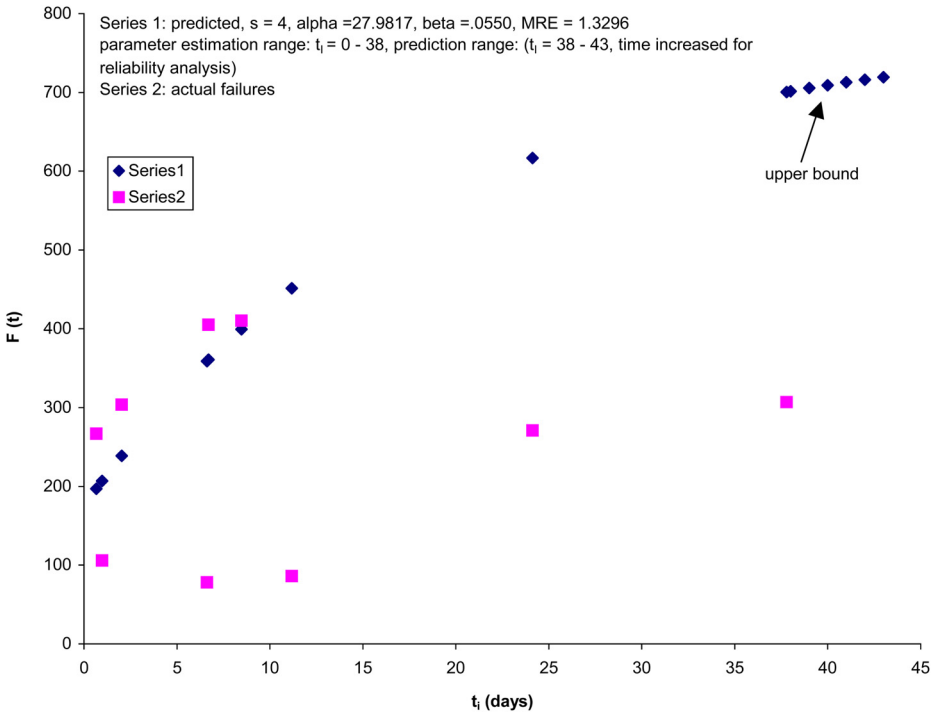


Fig. 8 NASA JM1 software: process cumulative failures F (t) vs. time in phase i t_i.

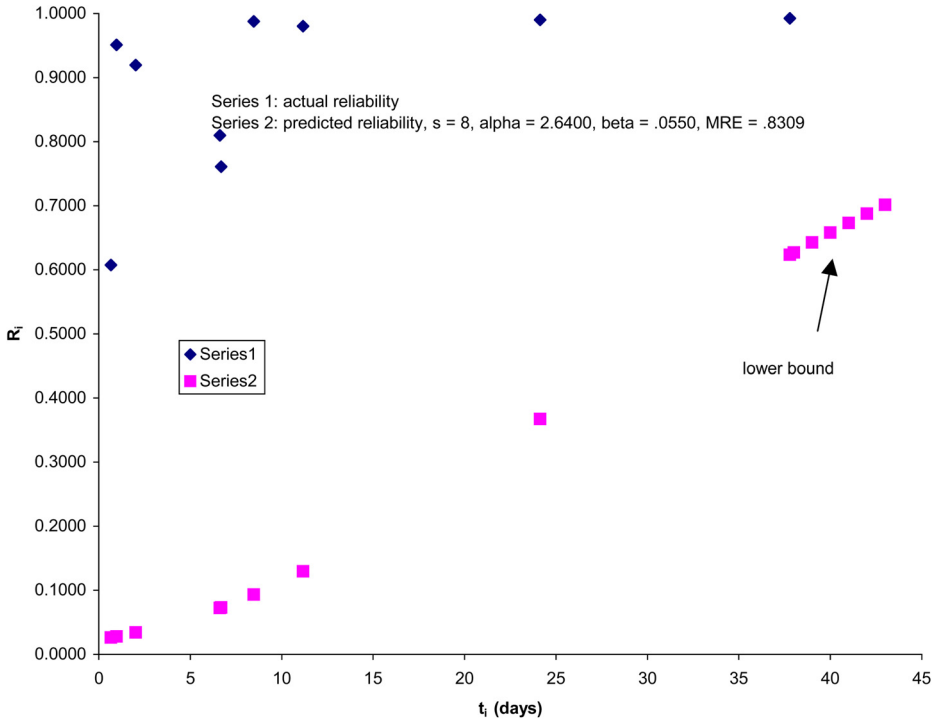


Fig. 9 NASA JM1: software: process reliability R_i vs. time spent in phase i (t_i).

In Fig. 5 we address the problem of identifying errant modules. We screen them by showing whether module reliability is below the reliability limit. In this case there are three such modules but fortunately reliability growth is demonstrated across modules.

Figure 6 demonstrates that while reliability growth may be demonstrated by individual modules, there can still be process reliability problems. In this case there are four phases below the reliability limit, necessitating a review of these processes by software engineering management.

In Fig. 7 we see the situation mentioned earlier of the reliability prediction approach using SMERFS (Series 3) achieving better prediction accuracy for product reliability, particularly during early values of t. This result points up the fact that it is wise to use more than one model when making software reliability predictions so that we can see whether one prediction would confirm another. For example, in the prediction range, the two approaches yield essentially the same result, despite the fact that the parameter values are vastly different.

In Fig. 8 the things to notice are: 1) predicted process cumulative defects provides an upper bound on the actual defects (therefore, it is wise to use a prediction as a bound, even if prediction accuracy is not outstanding) and 2) the prediction range corresponds to additional time allocated to the reliability analysis phase, because this phase of software production is usually given short shrift. A similar principle is illustrated in Fig. 9. In this case the predicted process reliability provides a lower bound on the actual reliability. Thus we could use the prediction to give us the minimum time that must be spent in each phase.

III. Summary

1. Phase efficiency can vary over phases. Thus this information can be used to identify phases that are in need of process correction.
2. It is important to see whether correcting defects tracks time spent in phases. If this were not the case, it could indicate that the production process is not producing expected benefits.
3. Defect rate should be examined to determine how many production channels are required to meet module production quotas.

4. It is critical to analyze defect count removal in each phase to see whether there are phases falling behind in correcting defects.
5. Over all modules, we need to ascertain whether there is reliability growth and identify for further quality processing modules that do not meet this criterion.
6. We must identify process for process enhancement, phases that do not meet the reliability limit.
7. We must take care, when predicting module reliability, to use more than one model because different models produce different predictions; we should apply the most accurate model.
8. We can use process predicted cumulative failures to advantage in providing an upper bound to actual failures.
9. Similarly, predicted process reliability can provide a lower bound to the actual reliability.

References

[1]Scacchi, W., "Process Models in Software Engineering", J. Marciniak (ed.), Encyclopedia of Software Engineering, 2nd, Edition, John Wiley and Sons Inc., New York, February 2001.

[2]Keller, T. and Schneidewind, N. F., "A Successful Application of Software Reliability Engineering for the NASA Space Shuttle", Software Reliability Engineering Case Studies, International Symposium on Software Reliability Engineering, November 3, Albuquerque, New Mexico, November 4, 1997, pp. 71–82.

[3]Jensen, P. A., Operations Management/Industrial Engineering, Internet, 2004.

[4]Krause, P., Freimut, B., and Suryn, W., "New Directions in Measurement for Software Quality Control," step, p. 129, 10th International Workshop on Software Technology and Engineering Practice, 2002.

[5]Monks, J. G., Operations Management, 2nd Edition, McGraw-Hill, 1996.

[6]Turner, W. C., Mize, J. H., and Nazemetz, J. W., Introduction to Industrial and Systems Engineering, 3rd Edition, Prentice Hall, 1993.

[7]Kirk, D., "A Flexible Software Process Model," ICSE, pp. 57–59, 26th International Conference on Software Engineering (ICSE'04), 2004.

[8]Miller, S. D., DeCarlo, R. A., and Mathur, A. P., "Modeling and control of the incremental software test process", Computer Software and Applications Conference, 2004. COMPSAC 2004. Proceedings of the 28th Annual International, Vol. 2, 28–30 September 2004, pp. 156–159.

[9]Cangussu, J. W., Karcich, R. M., Mathur, A. P., and DeCarlo, R. A., "Software Release Control using Defect Based Quality Estimation," issue, pp. 440–450, 15th International Symposium on Software Reliability Engineering (ISSRE'04), 2004.

[10]Cangussu, J.W., DeCarlo, R.A., and Mathur, A.P., "Using sensitivity analysis to validate a state variable model of the software test process", IEEE Transactions on Software Engineering, Vol. 29, No. 5, May 2003, pp. 430–443.

[11]Schneidewind, N. F., "Reliability Modeling for Safety Critical Software", IEEE Transactions on Reliability, Vol. 46, No.1, March 1997, pp. 88–98.

[12]Schneidewind, N. F., "Predicting shuttle software reliability with parameter evaluation" Innovations Systems Software Engineering, Springer-Verlag London Limited 2007.

[13]Farr, W. H., and Smith, O. D., *Statistical Modeling and Estimation of Reliability Functions for Software (SMERFS) Users Guide*, NAVSWC TR-84-373, Revision 2, Naval Surface Warfare Center, Dahlgren, Virginia.

[14]Schneidewind, N. F., "Experience Report on Using Object-Oriented Design for Software Maintenance", Journal of Software Maintenance and Evolution, (Wiley InterScience), 7 June 2007, Vol. 19, No. 3, pp. 183–201.

Michael Hinchey
Associate editor