Vol. 35 No. 12 Transactions of Information Processing Society of Japan Dec. 1994

Regular Paper

A New Software Reliability Growth Model Predicated on Counting Processes for Instruction Execution

MASAMI NORO,[†] KUNIO GOTO[†] and KATSUSHIGE SAWAKI[†]

We have developed a new software reliability growth model based on counting processes for instruction execution in software. Our model defines the reliability of the program at time t, $R(t)$, as

$$R(t)=\prod_{i\in S} exp\left[-\frac{p_i}{a+p_i}\lambda q_i t\right]$$

where

$S=\{G, L, C, O\}$, G, L, C, O denotes global, local, communication, and others, respectively;
a is the total number of faults;
p_i is the probability that failure occurs at the first execution of a class-i instruction, proportional to the number of faults in class-i instructions (we call p_i *the initial probability of failure*);
λ is the rate of instruction execution; and
q_i is the probability that an instruction execution is of class-i.

Through analysis using the proposed model, we conclude that higher software reliability can be achieved with data abstraction techniques than with functional decomposition, under reasonable assumptions. We note also that the exponential NHPP model is a special case of our theoretical model, and that the results of our model therefore agree with those of the NHPP model.

1. Introduction

A number of software reliability growth models (SRGMs) have been proposed to measure quantitatively the reliability of large-scale software through statistical analysis of defect behavior.[3),5),7),9)-12)] In these traditional models, software is treated as a black-box entity, and its defect behavior is only macroscopically observed. In general, and in particular for non-homogeneous Poisson process (NHPP) models, the following characteristics hold:

1) Formulas are based on a process that counts overall failures of target software.
2) Different types of failure are not distinguished.

These characteristics of NHPP models ensure simplicity, and, as a result, the models can be easily applied to real projects. However, the internal structure of software should be considered in order to evaluate the reliability of various types of software, or the reliability of software written by using different modern techniques. Therefore, microscopic models that represent internal software structure are needed.

The software engineering community has developed several productive principles and paradigms over the last decade. In particular, the object-oriented paradigm has been a great success, and has been successfully applied to real projects.[4)] One of the most important principles of the object-oriented paradigm is the concept of data abstraction.[6)] The *object* is the construct employed for data abstraction. Since it protects its internal data from illegal or undesirable accesses from outside, the data abstraction technique is said to enhance software reliability. This leads to one of the most important ideas of the object-oriented paradigm: well-structured software, which is designed as a set of abstract data, has high reliability.

Traditional SRGMs measure the quality of software structure in terms of the total number of faults, which is one type of estimated parameter. Unfortunately, the cause-and-effect relationship between the structure and the reliability of the target software is not captured by the estimation of other parameters, because the models are formulated from a bird's eye view. In other words, while it is possible to discuss how reliable the software is, there is no way to know how much the integrity of the code's structure contributes to reliability. Thus a major application of traditional SRGMs has been to estimate

† Department of Information Systems and Quantitative Sciences, Nanzan University

the total number of faults in the target software.

If we develop an SRGM that represents the internal structure of the software in detail, the values of various parameters estimated from the defect behavior will naturally yield measures for the quality of the target software structure. With such a model, we can analyze the causes and types of faults, in addition to estimating the total number of faults. Therefore, the model can be more actively used to control the software development process, as demonstrated by Yamada and Takahashi,[13] and by Basili's TAME.[1]

Following the discussion above, we propose a new SRGM[8] that represents the structure of the target software. Our model has, in concrete terms, the following features:

1) Instruction execution is assumed to be a counting process.
2) The instructions in the code are categorized into several classes.
3) A set of parameters for the model can be used to measure the quality of the target software structure.

In the case where the instruction execution obeys a Poisson process, let us preview a formula obtained in the next section for the reliability of the software at time t, $R(t)$:

$$R(t)=\prod_{i\in S}\exp\left[-\frac{p_i}{a+p_i}\lambda q_i t\right]$$

where

$S=\{G, L, C, O\}$, G, L, C, O denotes global, local, communication, and others, respectively,

a is the total number of faults,

p_i is the probability that failure occurs at the first execution of a class-i instruction, proportional to the number of faults in class-i instructions (we call p_i *the initial probability of failure*),

λ is the rate of instruction execution, and

q_i is the probability that an instruction execution is of class-i.

Using this formula, we compare the reliability of software designed by data abstraction with that of software designed by functional decomposition. We conclude that the former is generally more reliable.

The rest of the paper is organized as follows. Section 2 describes the proposed SRGM and justifies the formula above. In Section 3, the reliability of software designed by data abstraction is compared with software designed by functional decomposition. We discuss the validity of our SRGM in Section 4.

2. A Software Reliability Growth Model Based on Instruction Execution

In this section, we develop an SRGM for sequential-processing software in two steps:

1) We derive a reliability function of the given time t for a single class of instructions, as in traditional models.
2) We categorize the instructions into several classes to represent the structure of the code, and then extend the model to the general case.

Throughout the paper, we make the following two assumptions:

Once a failure occurs, it is always detected.

A single failure is always caused by a single fault.

That is, an event of a failure occurrence is equivalent to that of fault detection.

2.1 The Basic Model

In general, an SRGM yields a continuous-time formula. Since failure may take place at any moment of time, it is natural to build our SRGM as a continuous-time model. First we focus on the series of time instants of instruction execution and derive a discrete-time formula. Then, using a Poisson process, we translate the discrete-time formula into the corresponding continuous-time formula for the case in which instruction execution may occur at random.

2.1.1 Instruction Execution and Failure

Figure 1 illustrates the basic assumptions of the relationships among instruction execution, occurrence of failures, and the probability of failure occurrence. Let us define a random variable, O_i as follows:

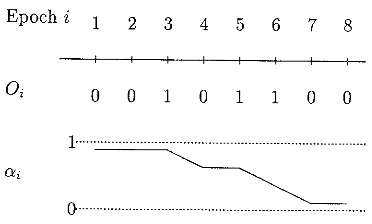


Fig. 1 Relationship between instruction execution, failure occurrence, and its probability in software.

$$O_i=\begin{cases}1 & \text{if a failure occurs and is detected at the epoch } i\,;\\ 0 & \text{otherwise}\end{cases}$$

where the epoch i is the time instance of the i-th execution of instructions. α_i represents the probability of failure occurrence at the epoch i, and its value decreases as the reliability grows. It is assumed that the probability of failure occurrence depends solely on, and is directly proportional to, the number of remaining faults.

The figure assumes the following scenario:

1) In epoch 3, a failure occurred, and therefore ($O_3=1$) was detected. The fault that caused the failure was detected and corrected, and the total number of faults was reduced by 1. The probability of failure occurrence α_3 was reduced.
2) In epochs 5 and 6 failures also occurred, the corresponding faults were corrected, and the probability was reduced.

2.1.2 The Mean Value Function for Failures in Discrete Time

Let p be the initial probability of failure occurrence, that is,

$$p=Pr\{\text{One of faults in the code is executed}\}$$
$$=\frac{\text{(the total number of faults)}}{\text{(the total number of instructions)}}\times\text{(constant)}.$$

Then, α_i is defined as:

$$\alpha_i=\frac{p\cdot E[\text{the number of remaining faults in the code at the epoch } i]}{\text{(the total number of faults in the code)}}$$
$$=\frac{(a-H_i)}{a}\cdot p \qquad (0)$$

where a is the total number of faults in the code. The mean value function, H_i, is

$$H_i=E[\text{the number of faults detected up to the epoch } i \text{ (the number of faults corrected through the epoch } i-1)].$$

To derive H_i, let $Q_i(k)$ be a probability mass function:

$$Q_i(k)=Pr\{\text{the number of faults detected through the epoch } i-1=k\}.$$

Then, the expectation of k becomes H_i:

$$H_i=\sum_{k=0}^{\infty}k\cdot Q_i(k) \quad \text{(the mean value with respect to } k).$$

Note that we have the following recurrence equation, $Q_i(k)$:

$$\begin{aligned}Q_i(k)&=Pr\{(k \text{ up to the epoch } i-1)\wedge(\text{no new failure occurs and no new fault is detected in the epoch } i-1)\}\\&\quad+Pr\{(k-1 \text{ up to the epoch } i-1)\wedge(\text{a new failure is detected in the epoch } i-1)\}\\&=Pr\{\text{no new failure occurs and no new fault is detected in the epoch } i-1 \mid k \text{ up to the epoch } i-1\}\times Q_{i-1}(k)\\&\quad+Pr\{[\text{a new failure occurs and a corresponding new fault is detected in the epoch } i-1]\wedge[k-1 \text{ up to the epoch } i-1]\}\times Q_{i-1}(k-1)\\&=(1-\alpha_i)\cdot Q_{i-1}(k)+\alpha_i Q_{i-1}(k-1),\end{aligned} \qquad (1)$$

where $i=1,2,3,\cdots$, $k=0,1,2,\cdots,a\wedge(i-1)$, and $Q_i(k)=0$ otherwise. Note that we assume that once a failure occurs, the corresponding fault which causes the failure, and which is always single, is always detected and fixed. That is, to say that no new failure occurs means that no new fault is detected.

Let $Q_i^*(z)$ be a probability-generating function $Q_i(k)$. That is,

$$Q_i^*(z)=\sum_{k=0}^{\infty}z^kQ_i(k).$$

Equation (1) becomes

$$\begin{aligned}Q_i^*(z)&=(1-\alpha_i)Q_{i-1}^*(z)+\alpha_i zQ_{i-1}^*(z)\\&=\{(1-\alpha_i)+\alpha_i z\}Q_{i-1}^*(z)\end{aligned}$$
$$Q_i^*(z)=\prod_{n=2}^{i}\{(1-\alpha_n)+\alpha_n z\}Q_1^*(z).$$

Note that $Q_1(0)=1$ and $Q_1^*(z)=1$; then,

$$Q_1^*(z)=1$$
$$Q_i^*(z)=\prod_{n=2}^{i}(1-\alpha_n+\alpha_n z). \qquad (2)$$

We have H_i in terms of α_n:

$$H_1=0$$
$$\begin{aligned}H_i&=\sum_{k=0}^{\infty}k\cdot Q_i(k)=\lim_{z\to1}\frac{dQ_i^*(z)}{dz}\\&=\lim_{z\to1}\sum_{k=2}^{i}\left\{\alpha_k\prod_{m=2,m\neq k}^{i}(1-\alpha_m+\alpha_m z)\right\}\\&=\sum_{n=2}^{i}\alpha_n \quad (i\geq2).\end{aligned} \qquad (3)$$

From equation (0),

$$\alpha_1=p$$
$$\alpha_i=p\cdot\frac{a-H_i}{a}$$

$$=p-\frac{p\sum_{n=2}^{i} a_n}{a} \quad (i\geq 2). \tag{4}$$

Now we take the difference as:

$$a_{i+1}-a_i=-\frac{p}{a}a_{i+1}$$

$$a_{i+1}=\frac{1}{1+\frac{p}{a}}a_i.$$

Hence,

$$a_i=\frac{p}{\left(1+\frac{p}{a}\right)^{i-1}}.$$

Again, from (0),

$$H_i=a\cdot\left\{1-\frac{1}{\left(1+\frac{p}{a}\right)^{i-1}}\right\} \quad (i\geq 1).$$

Since $\frac{p}{a}>0$,

$$\lim_{i\to\infty} a_i=0$$

$$\lim_{i\to\infty} H_i=a.$$

The derivation so far does not depend on any specific probability distribution of consecutive instruction executions. Again, note that H_i is a mean value function of not only faults but also failures, on the basis of our assumption.

2.1.3 The Continuous-Time Formula

If we assume that the executions of software instructions are carried out according to a Poisson process with rate λ, we have a quite simple form for the reliability function. Since a Poisson process is known to represent exponential interarrival time at a service facility (in terms of queuing theory), this assumption is reasonable in most cases of instruction execution in software.

$$Pr\{\text{the number of instructions executed in time interval } t=i\} = \frac{(\lambda t)^i}{i!}e^{-\lambda t}.$$

Thus, the mean value function of failures at a given time t is

$$H(t)=\sum_{i=0}^{\infty} H_{i+1}\cdot\frac{(\lambda t)^i}{i!}e^{-\lambda t}.$$

Hence,

$$H(t)=a\left(1-exp\left[-\frac{p}{a+p}\lambda t\right]\right).$$

The definition of failure rate is

$$d(t)=\frac{\frac{dH(t)}{dt}}{a-H(t)}.$$

Then, substituting $H(t)$ in the expression, we have

$$d(t)=\frac{p}{a+p}\lambda.$$

Finally, we have the reliability, $R(t)=exp\left[-\int_0^t d(\tau)\,d\tau\right]$:

$$R(t)=exp\left[-\frac{p}{a+p}\lambda t\right].$$

2.2 Extension of the Basic Model for Multiple Classes of Software Instructions

We now classify instructions into the following four categories:

1. Global data access
2. Local data access
3. Inter-module communication
4. Others

Here, a module is defined as a set of subprograms (procedures or functions) and/or data definitions. A software system consists of multiple modules. Inter-module communication is defined as follows:

> When an instruction in module M_1 is a call to a subprogram p in module M_2, inter-module communication occurs between M_1 and M_2, and is defined to be the set of actions that are the parameter passings of p's invocation, along with the return value in the case of a function call.

Let p_i be the initial probability of failure occurrence, and let q_i be the probability of instruction execution, where suffixes such as G, L, C, O mean global data access, local data access, inter-module communication, and others, respectively. Say p_G denotes the initial probability of failure occurrence in global data access. If we assume the executions of the different types of instructions to be independent of one another, the failure rate can be obtained:

$$d(t)=\sum_{i\in S}\frac{p_i}{a+p_i}\lambda q_i$$

where

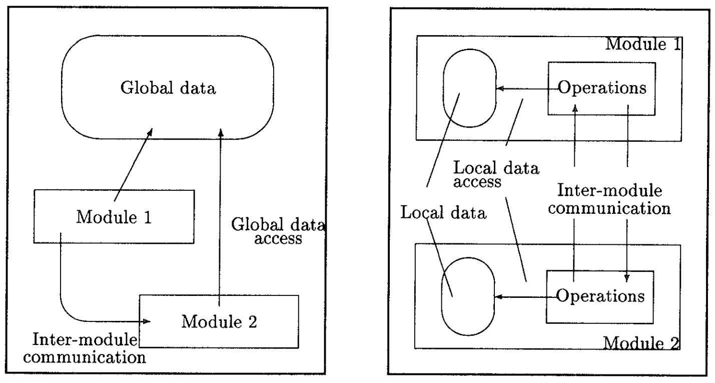
$$S=\{G, L, C, O\}.$$

Reliability is defined as

$$R(t)=\prod_{i\in S} exp\left[-\frac{p_i}{a+p_i}\lambda q_i t\right].$$

3. Data Abstraction versus Functional Decomposition from the Perspective of Our Model

This section discusses the reliability of *data*



(a) Functional Decomposition Software (b) Data Abstraction Software

Fig. 2 Structure of software.

abstraction software and *functional decomposition software*☆. We use the term *data abstraction software* for software designed by using the abstract data type technique,[6] or an object-oriented design method such as Booch's.[2] Software developed by using the design method of functional decomposition is called functional decomposition software. The sets of parameters for both types of software are defined according to the SRGM proposed in the previous section. Finally, their reliabilities are compared under several well-accepted assumptions.

3.1 Data Abstraction Software and Functional Decomposition Software

The structure of functional decomposition software and data abstraction software is illustrated in **Figs. 2**(a) and (b), respectively. As shown in the figure, functional decomposition software consists of a set of global data and modules that are sets of subprograms. Data abstraction software, on the other hand, is composed of modules that have internal local data and operations (procedures and functions) for accessing the internal data. No global data exist in data abstraction software.

3.2 Which Is More Reliable?

Let p_i^j be the initial probability of failure occurrence and let q_i^j be the probability of instruction execution, in which $i \in S = \{G, L, C, O\}$ and $j \in T = \{d, f\}$. G, L, C, and O denote global, local, communication and others, as before. Let d denote *data abstraction software* and let f denote *functional decomposition software*. For example, p_G^f is the initial probability of failure occurrence at the time of global data access in functional decomposition software.

Let $d_d(t)$ and $d_f(t)$ show the failure rates of data abstraction software and functional decomposition software at time t. That is,

$$d_d(t) = \sum_{i \in S} \frac{p_i^d}{a + p_i^d} \lambda q_i^d$$

$$d_f(t) = \sum_{i \in S} \frac{p_i^f}{a + p_i^f} \lambda q_i^f.$$

Assumptions Made for Comparison

The following assumptions are made for comparison of these two types of software:

1. Assumptions about the probability of instruction execution:
   1a) Data abstraction software does not have global data in its components.
      It is assumed that the data structure is completely hidden in each module of data abstraction software.
   1b) Functional decomposition software has global data that are accessed by multiple modules.

☆ We do not want to draw the general conclusion that *data abstraction techniques are always superior to functional decomposition design methods in all cases*. Nor do we insist that the techniques and methods cannot be combined. Our intention is to show that information-hiding realized by data abstraction techniques enhances software reliability. To make the discussion clearer, we made extreme assumptions that information-hiding is achieved in full in data abstraction software and is not achieved at all in functional decomposition software. In practice, data abstraction techniques can be incorporated in a design process that follows a functional decomposition method, and vice versa.

Table 1 Initial probability of failure occurrence and probability of instruction execution.

	Global data access	Local data access	Communication	Others
Initial probability of failure occurrence	$p_G^d=0$ p_G^f	$p_L^d=p_L^f$ $p_L^f=p_L^d$	$p_C^d=p_C^f$ $p_C^f=p_C^d$	$p_O^d=p_O^f$ $p_O^f=p_O^d$
Probability of instruction execution	$q_G^d=0$ $q_G^f=q_C+q_L$	$q_L^d=q_L+q_L^f$ q_L^f	$q_C^d=q_C+q_C^f$ q_C^f	$q_O^d=q_O^f$ $q_O^f=q_O^d$

Since we assume that data abstraction is done only in a limited way in functional decomposition software, its global data are accessed by multiple modules.

1c) Functional decomposition software realizes a part of the inter-module communication in data abstraction software through data flow via global data.

Global data in functional decomposition software can thus be categorized into two types: the first type due to incomplete abstraction; and the second type due to data flow via the data.

2. Assumptions about the initial probability of failure occurrence:

2a) The initial probability is 0 if it is for instructions never executed ($p_G^d=0$ because $q_G^d=0$.)

2b) The initial probabilities of failure occurrences in local data access, inter-module communication, and others are the same.

We assume that both types of software are designed and coded by programmers with the same level of skill. Hence, there is an equal chance of faults being introduced in code for these three types of instructions. As a result, all of their initial probabilities will be equal.

Table 1 summarizes, in accordance with the above assumptions, the initial probability of failure occurrence and the probability of each instruction being executed. In the table, q_C represents the probability of data flowing via global data, and q_L represents the probability of accessing global data that have been declared as a result of incomplete data abstraction in functional decomposition software.

Let Δd be the difference between the failure rates of data abstraction software and functional decomposition software; that is,

$$\Delta d=d_d(t)-d_f(t).$$

$$\begin{aligned}\Delta d&=\sum_{i\in S}\frac{p_i^d}{a+p_i^d}\lambda q_i^d-\sum_{i\in S}\frac{p_i^f}{a+p_i^f}\lambda q_i^f\\&=\frac{p_L^d-p_G^f}{(a+p_L^d)(a+p_G^f)}a\lambda q_L\\&+\frac{p_C^d-p_G^f}{(a+p_C^d)(a+p_G^f)}a\lambda q_C.\end{aligned}\tag{5}$$

The denominator of each term of equation (5) is positive, and the parameters a, λ, q_L and q_C also may take only positive values. The numerators, therefore, determine the sign of Δd. The value of Δd is always negative, for the following reasons:

1. $p_G^f>p_L^d$, because the possibility of faults being introduced into instructions that access local data is smaller than in code employing global data access. That is, abstracted data are decomposed into smaller chunks than global data, and these chunks are protected from illegal access from outside. Hence, faults are easily introduced into instructions that reference global data.
2. $p_G^f>p_C^d$, because faults are more likely than in inter-module communication to be introduced in code that has data flow via global data, which makes use of side-effects.

In accordance with the above the discussion, we can objectively draw the following conclusion:

The data abstraction technique contributes to the achievement of highly reliable software.

4. Validation of Our Model

Our Model and the NHPP Model

Our SRGM can be considered as a refinement of the exponential NHPP model[3] in which the mean value function of failures is represented as

$$H(t)=a(1-e^{-bt})$$

when we define b as

$$b=\frac{p}{a+p}\lambda.$$

That is, our SRGM explains the failure behavior of software more precisely than the exponential NHPP model does. The exponential NHPP model is, from real project experience, said to explain the failure behavior of large-scale software. From this fact, and the discussion in Section 3, we can draw the following conclusion:

Data abstraction is a useful technique for enhancing the reliability of large-scale software.

How Can We Demonstrate the Usefulness of Our Model?

The next task is to demonstrate the practicality of our SRGM. We plan to take the following general approach to demonstrate this:

1. Gather failure data from a real project.
2. Estimate the parameters of the SRGM.
3. Validate the model by matching the estimated values of the parameters (particularly the total number of faults) and the actual set of values observed.

At present, a data set from real projects (preferably from a third party) detailed enough to be applicable to our SRGM is not available. Meanwhile, we are attempting to validate the model by simulation. The steps in our simulation experiment are as follows:

1. Generate pseudo-failure data based on a Gonpertz curve.
2. Estimate the parameters of our SRGM, using the data generated.
3. Validate the method by comparing the estimated total number of faults with the value obtained in the first step.

5. Conclusion

In this paper, we have developed a new SRGM that represents the executions of software instructions as counting processes. Using this SRGM, we discussed whether data abstraction techniques contribute to the design of highly reliable software, and concluded that *they do*. This is shown formally under well-accepted assumptions in software development. That is, we have lent some *objective* support to well-known folklore existing among many good software developers.

Our future research will include the following topics:

1. Further refinement of the model.
   We made the assumption that a failure occurrence is equal to a fault detection. We will refine the model for the following weaker assumption:
   - A combination of multiple faults may cause one failure.
   - Failure occurrence is distinguishable from its detection.
2. Generalization of the model.
   In the process of our model formulation, described in Section 2, we assumed that instruction executions follow a Poisson process. More generalized distributions, such as non-homogeneous Poisson processes, can be used to formulate a more generalized model.
3. Derivation of a family of models.
   In the current form of our SRGM, we assume that α_i is proportional to the remaining faults, as in Section 2. A different expression for α_i would result in a family of models based on our model formulation framework. This is reflected in the fact that various NHPP models can be derived by defining the mean value function of failures in different ways. Thus, our SRGM is a meta-model in which the key parameter is α_i.
4. Demonstration of the usefulness of our SRGM by means of real project data.

References

1) Basili, V. R.: Software Development: A Paradigm for the Future, *Proc. COMPSAC '89*, pp. 471-485 (1989).

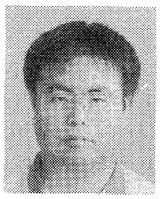
2) Booch, G.: *Software Engineering with Ada*, 2nd ed. Benjamin/Cummings Pub. Co., Menlo Park, CA (1987).

3) Goel, A. L. and Okumoto, K.: Time-Dependent Error Detection Rate Model for Software Reliability and Other Performance Measures, *IEEE Trans. Reliability*, Vol. R-28, pp. 206-211 (1979).

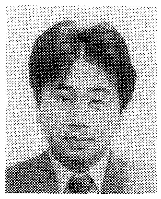
4) Harrison, W. H. and Sweeney, P. F.: Good News, Bad News: Experience Building a Software Development Environment Using the Object-Oriented Paradigm, *Proc. OOPSLA '89*, pp. 85-94 (1989).

5) Jelinski, Z. and Moranda, P.: *Statistical Computer Performance Evaluation in Software Reliability Research*, Freiberger, W. ed., Academic Press Inc., New York (1972).
6) Liskov, B. and Zilles, S.: Programming with Abstract Data Types, *SIGPLAN Notices*, Vol. 9, No. 4, pp. 50-59 (1974).
7) Musa, D. J.: The Measurement and Management of Software Reliability: How Good Are They and How Can They Be Improved ?, *IEEE Trans. Softw. Eng.*, Vol. SE-6, No. 9, pp. 489-500 (1980).
8) Noro, M., Goto, K. and Sawaki, K.: A Software Reliability Growth Model Predicated on Instruction Execution (in Japanese), *IPSJ SIG Notes*, Vol. 93-se-91, No. 16, pp. 41-48 (1993).
9) Ohba, M.: Software Reliability Analysis Models, *IBM J. Res. & Dev.*, Vol. 28, pp. 428-443 (1984).
10) Tohma, Y. et al.: Structural Approach to the Estimation of the Number of Residential Software Faults Based on the Hyper-Geometric Distribution, *IEEE Trans. Softw. Eng.*, Vol. SE-15, No. 3, pp. 345-355 (1989).
11) Yamada, S. and Osaki, S.: Software Reliability Growth Modeling: Models and Applications, *IEEE Trans. Softw. Eng.*, Vol. SE-11, No. 12, pp. 1431-1437 (1985).
12) Yamada, S.: Software Quality/Reliability Measurement and Assessment: Software Reliability Growth Model and Data Analysis, *J. Inf. Process.*, Vol. 14, No. 3, pp. 254-266 (1991).
13) Yamada, S. and Takahashi, M.: *Introduction to Software Management Model* (in Japanese), p. 254, Kyoritsu-Shuppan, Tokyo (1993).

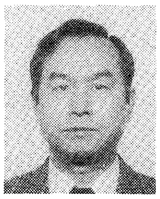
(Received February 28, 1994)
(Accepted September 6, 1994)



Masami Noro was born in 1958. He received B. E., M. E., and Ph. D. degrees in Administration Engineering from Keio University, Tokyo, Japan.
He is an associate professor of the Department of Information Systems and Quantitative Sciences, Nanzan University, Nagoya, Japan. His research interests include all aspects of software quality assurance, and designing distributed object-oriented programming system.
Dr. Noro is a member of IPSJ, JSSST, IECE, ACM, and IEEE.



Kunio Goto was born in 1957. He received the B. E., M. E., and D. E. degrees in Applied Mathematics and Physics from Kyoto University, Kyoto, Japan.
He is currently an associate professor of the Department of Information Systems and Quantitative Sciences, Nanzan University, Nagoya, Japan. His research interests are in computer communication networks and their applications and performance evaluation.
Dr. Goto is a member of the IEEE, ACM, IEICE and ORSJ.



Katsushige Sawaki was born in 1949 and is Professor at the Department of Information Systems and Quantitative Sciences, Nanzan University. He received Ph. D. from the University of British Columbia in 1975. His areas of research include dynamic programming, Markov decision processes and mathematical finance. His articles on these topics appear in J. Mathematical Analysis and Application, Naval Research Logistics Quarterly, Transportation Science and J. Operations Research Society of Japan among others.