Some Issues in Multi-Phase Software Reliability Modeling*

M. A. Vouk and K.C. Tai

Abstract

During early software testing phases, testing profiles are often very different from operational profiles. Consequently, assessment of operational software quality during these non-operational testing stages is difficult, and is open to interpretation. The paper discusses some issues related to this. Software is assumed to be a large system composed of components that evolve in parallel. The focus is on early identification of software components that in operation may be excessively error-prone. The approach involves definition of states based on static and dynamic propositions about the verification and testing history of the software, and the use of that information in models that span multiple testing phases. An example based on a risk model is presented.

1. Introduction

The use of software reliability engineering (SRE), in organizations with advanced software processes, is on the rise. But, some practical obstacles still remain.

For example, SRE requires testing based on an operational profile. An operational profile is a set of relative frequencies of occurrence of the operations associated with the software during its use in the field [Mus93]. Interpretation of many software reliability models assumes failure detection based on operational profiles [Mus87]. Since this assumption is usually violated during early software testing phases (for example, during unit-testing and integration-testing), assessment and control of software quality growth during non-operational testing phases is difficult and open to interpretation.

Another confounding factor can be the (necessary) discontinuities that different testing strategies introduce within one testing phase, or between adjacent testing phases. For instance, unit-testing concentrates on the functionality and coverage of the structures within a software unit; integration-testing concentrates on the coverage of the functions and links that involve two or more software units, etc. It is not unusual to observe an apparent failure-intensity decay (reliability growth) during one of the phases, followed by an upsurge in the failure-intensity in the next phase (due to different types of failures). This oscillatory effect can make reliability growth modeling difficult, although several different approaches have been suggested [e.g., Mus87, Lyu92].

In an organization that constructs its final deliverable software out of a number of components that evolve in parallel, an added problem can be the variability of the quality across system components.

The need to recognize these problems early, so that appropriate corrections can be undertaken within one software release frame, is obvious. How to achieve this is less clear. In general, it is necessary to link the symptoms observed during the early testing phases with the effects observed in the later phases, such, as early operational phase. Several authors have published models that attempt to relate some early software metrics, such as, the size of the code, Halstead length, or cyclomatic number, to the failure proneness of the program [Mun92, Bri93].

This paper is concerned with some issues related to the use of software reliability engineering (SRE) indicators available during early software testing of a large multi-component software system to:

i) **Quantify component quality** expressed, for example, as the number of failures (or problem reports) expected during initial operational deployment of the component;

ii) **Identify problem-prone software components**, that is, components that might show an increased propensity to...
failure during initial operational deployment (e.g., number of field problem reports); iii) **Guide software testing process** to minimize the number of failures that can be expected from the components in the future phases of their life cycle.

The software system model we consider here is a system that consists of a collection of software products, or software components, arranged in a certain hierarchy of usage. The components that support the basic functionalities of the system form the system basis, or root. The interaction among different components and their hierarchy can be described using the system call tree structure [Mus93].

The issues presented in this paper are based, in part, on our experiences with several large commercial systems which, for proprietary reasons, are not identified. The results are exploratory in nature, and are intended to continue the dialogue and provide incentive for further research and input on these topics.

Section 2 discusses the SRE metrics that may be of interest. Section 3 presents a simple multi-phase model for problem identification. The model serves to illustrate the approach, but needs to be extended and made more robust. Section 4 discusses the role of reliability growth models during non-operational testing and the possibilities this offers in terms of software process control.

2. Metrics

We distinguish: a) metrics collected in order to quantify the current software process and quality, and b) the parameters that describe the quality of software in later stages, and that are estimated using the collected data. We call the first group "process and product quality descriptors", or input **drivers**, and the latter "(future state) quality estimators", or **estimators**. We illustrate the metrics through several example drivers and estimators.

2.1 Estimators

2.1.1 Number of Failures

A metric that may be readily available in many organizations is the number of failures observed per product, or component. The number of failures may be chosen because:

i) It has intuitive relationship with the quality of software. For example, one should be uneasy about the software that exhibits more than a certain number of problems immediately upon release to its operational environment.

ii) It has practical value in terms of the amount of attention (and work) the software will receive due to the reported problems.

The number of observed problems will depend on how often and in what manner the system, or a component, is used. This suggests that the data also needs to be collected on the software usage, size, and any other relevant metric. Distinction needs to be made between unique failures, repeated failures, and the underlying faults [Mus87].

2.1.2 Failure-intensity

Failure-intensity is a classical SRE metric [Mus87]. It can be defined as the rate of change of the mean value function, or the number of failures per unit time. The mean value function is the average cumulative number of failures at a point in time. In the context of the failure-intensity, "time" is the exposure time of software to use, or to testing. It can be the CPU time, the calendar time, the expended testing or debugging time (or effort), and even the number of test cases run.

For example, we distinguish time-oriented failure-intensity (that is, failures per unit time), and test-case intensity (that is, number of failures per test case). In the case of operational testing, an excellent exposure metric is the CPU execution time [Mus87, Jon91, Cra92], but this may not be the case in a non-operational testing environment.

In addition to the instantaneous (classical) failure-intensity, we have found it useful to compute the following three failure-intensity metrics: the average failure-intensity expressed in terms of **failures per unit time** (total failures over total time); "final failure-intensity", which is the failure-intensity expressed in terms of failures per unit time averaged over the last 10% to 20% of the reported testing; and "test-case failure-intensity", which is average failure-intensity expressed in terms of **failures per test case**, or the number of unique failures per unique test case.

2.2 Drivers

A number of factors may influence the observed field quality of software. For example,
the quality of the verification, validation and testing efforts, the frequency of usage of the component, the size of the changes made in the software, and the number and location of the residual defects.

In order to relate the drivers and the estimators, it is necessary to find either a direct analytical, or a tabulated, relationship which connects many different, possibly continuous, levels of conditions (or causes) with the final effect (reliability). The approach where the input conditions are discrete states requires determination of threshold values for these variables (or metrics). When an input metric exceeds (or is below, depending on the metric) a threshold, transition to the next state of quality may take place.

2.2.2 Usage

The more often the product is used, the more likely it is that defects will be found in it. A full implementation of SRE requires determination of operational profile(s) [Mus93], and analysis of observed problems in that context. For example, it should be established whether the cases showing large numbers of reported problems owe that to very frequent usage of a component that has an average residual fault density, or to an excessive residual fault density in a product that is being used at the rate typical for most other products.

Definition and use of the appropriate operational profile(s) is essential for accurate evaluation of the testing process and its effects. Lacking actual operational usage information, it may be possible to estimate it. One possibility is to use the dynamic operational deployment information and figures [Mus93]. Another, less accurate way, is to statically analyze the product component call graphs. The call graph is a tree-like structure which describes the interactions between different product components and their hierarchy [Mus93]. In that context we consider two metrics. The component "Importance" and the component "Call Usage".

The "Importance" metric attempts to indicate the importance, and, indirectly, possible usage frequency of a component (or function). The "level 1", or root, components are most important and are likely to be most frequently used because all components above them use them to some degree. The set of components that connect directly to the this level are "level 2" components. The level of a component can be computed by counting the number of graph edges that exist between the node (component) of interest and "level 1", and adding 1 to that count.

A more accurate description of the static usage frequency might be the count of the total number of components that use a component. With that in mind we constructed the metric we named "Call Usage". This metric is "1 plus the..."
total number of distinct siblings that can call a given component on the system call-tree. The 1 is added to account for the execution (self-use) of the component itself.

![Figure 2.2](image)

**Figure 2.2** The number of operational failures may correlate with the total number of product call-tree siblings.

Figure 2.2 illustrates a possible relationship between the number of failures, and the usage level. The entries marked with a large circle illustrate components that may have experienced high levels of change (for example, over 10,000 lines of code). The horizontal axis is the logarithm of "Call Usage", and the vertical axis is the logarithm of the total number of field problems reported during, for example, an early operational period lasting X units of time. We see that, in this illustration, high "Call Usage" components show increased incidence of problem reports, and may have experienced a higher change level.

![Figure 2.3](image)

**Figure 2.3** Possible relationship between the number of failures and the "Testing Effort Coverage".

The assumption, made above, that each sibling contributes equally to the load on the root node is not likely to be entirely true in practice, and this may introduce distortions into parameter estimates. The contribution may vary depending on the sibling, platform, deployment site, customer, user, etc.

### 2.2.3 Effort

The usage information, although quite helpful, may be incomplete and not adequate as far as prediction of the problem-proneness of components is concerned. For example, usage usually does not reflect prior knowledge about the component evolution, such as its change level, or verification and testing history.

A driver that attempts to combine two historical variables is the "Testing Effort Coverage". It is meant to be an indicator of the effort invested into testing. The larger the effort, the smaller would be the residual number of faults one would expect. The general metrics is the "number of unique test cases per unit of change". The particular metric may be the "number of unique test cases per changed line of code", or the "number of test cases per decision node". The assumption is that the testers select the test cases based on some internal importance criteria related to the operations a component performs, and that they attempt to "cover" as many of the elements (or functions), that make up the component, as possible. The metric implies that there may be a threshold which would indicate the effectiveness, or the quality, of testing and, perhaps, guarantee a level of freedom from errors of a certain type.

A possible relationships is shown in Figure 2.3. We see that an analysis of this type can yield target values for the "Testing Effort Coverage". For instance, components that have not received at least one test case per 10 to 20 lines of changed code (0.1 to 0.05 test cases per line of code) may show problems in the later phases. In our experience, the false alarm rates for this type of metric alone may be of the order of 30-40%. Although, the "Testing Effort Coverage" appears to be an indicator of the problem-potential of a component, its false alarm rate may be unacceptably high, and, for practical use, it may need to be combined with other metrics.

### 2.2.4 Failure-intensity

Failure-intensity can also serve as a driver. However, our experience is that in the context of
non-operational testing, classical time-based failure-intensity appears to have a very limited meaning. To be useful, the failure-intensity must be associated with the particular testing strategy and effort used, the coverage of the product operations and functionalities, and, if possible, it must be corrected for the distortions induced by the non-operational nature of the testing profile.

Part of the problem may stem from the fact that, during component and integration testing, different components of a software system may execute at different speeds (for example, they may run on different processors). This may mean that different test cases may use different amounts of execution time, and, unless corrections are applied, a typical integration test-suite for one component may accumulate considerably more time per test case, than a similar test-suite for another component of the system. Normalization of the failure-intensities across different components may be necessary [Mus87]. One possibility is to use instantaneous or average "test-case failure-intensity", which is expressed in terms of failures per test case, or the number of unique failures per unique test case.

3. Risk Modeling

This section addresses the use of early testing information in identification of components that may be problem-prone in the field. Several approaches have been proposed [Kho90, Mun92, Bri93]. Highly correlated nature of the early software verification and testing events may require the use of a more sophisticated, time-series, approach [Sin92]. We illustrate some of the issues through a risk model [Ehr85, Boe89].

3.1 Process States

At the end of a non-operational testing phase an indication of the state of the software may be available as a set of conditions. In the most general form, the conditions will be compound, and will reflect the evolution process and history of software. The conditions will derive from quantification of a number of factors that may influence the operational quality of software. For example, the quality of the verification, validation and testing efforts, the operational usage profile, the size of the changes, and the number and location of the residual defects.

What is of interest here is the likelihood that the current state will lead to an unsatisfactory final state. For example, a state where the component exhibits high failure rate during its operational use. A specific state may be defined by providing a set of propositions (e.g., conditions that are true) that describe the achieved characteristics of the software, and its progress through the development process. For example,

\[ P_i(S_f | M_1 > m_1, M_2 > m_2, M_3 = m_3, \ldots M_n > m_n) \]

might indicate the conditional probability that state, \( S_i \), will lead to failure (for example, state \( S_f \) which is described by the condition: "initial operational failure intensity is more than 0.000006 failures per minute"). This transition probability is conditioned on the truth of propositions for \( M_1 \) through \( M_n \). These propositions capture the development history, the process, and, possibly, the operating environment. They indicate the conditions that have been met during the verification and testing process. Some of the metrics derive from the past, some from the current, and some from the expected future phases. For example, \( M_3 \) may be "Coding Inspection Effort Intensity" [Chr90], \( M_2 \) may be the current "Testing Effort Coverage", and \( M_1 \) may be the "Call Usage". Of course other drivers should also be considered.

Each identified state should be evaluated for the probability that it occurs, \( P(S) \), and for its capability to indicate (influence) what the future state of the process could be (for example, by its a posteriori probability, \( P_i(S_f | S) \)). This information can be used to define risk models.

When defining states and computing probabilities we need to account for the fact that many conditions are not mutually independent by considering their joint probabilities. We illustrate the issues using the following conditions (events):

i) A component has experienced high level of change.

ii) A component has high usage level.

iii) A component has not been "covered" with sufficient number of test cases.

iv) A component exhibits high test-case failure-intensity.

v) A component has not been "covered" with sufficient number of test cases and it exhibits high test-case failure-intensity.
Table 3.1 A comparison of the indicator metrics with respect to their capability to identify problem-prone products

<table>
<thead>
<tr>
<th>Input Variable</th>
<th>Change Level</th>
<th>Importance Level</th>
<th>Test Cases per Changed Line of Code</th>
<th>Unique Failures per Unique Test Case</th>
<th>UF/UTC and TC/ULOC</th>
<th>(8-level) * UF/UTC and TC/ULOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of Components</td>
<td>N</td>
<td>19</td>
<td>19</td>
<td>17</td>
<td>17</td>
<td>17</td>
</tr>
<tr>
<td>Total number that is considered Problem-Prone (i.e., in state Sf)</td>
<td>NP</td>
<td>P(Sf) = NP/N</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>Condition (C)</td>
<td>&gt; 10,000 LOC</td>
<td>≤ 3</td>
<td>≤ 2/25</td>
<td>&gt; 1/25</td>
<td>UF/UTC &gt; 0.04 and TC/UPS &lt; 0.08</td>
<td>(8-level)* UF/UTC &gt; 0.15 and TC/UPS &lt; 0.08</td>
</tr>
<tr>
<td>Number of Components Satisfying the Condition</td>
<td>NC</td>
<td>P(C) = NC/N</td>
<td>9</td>
<td>9</td>
<td>13</td>
<td>16</td>
</tr>
<tr>
<td>Number of Problem-Prone Components Satisfying the Condition</td>
<td>NPC</td>
<td>8</td>
<td>7</td>
<td>9</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>Number of Problem-Prone Components NOT Identified</td>
<td>NP-NPC</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Number of Components MIS-Identified as Problem Prone (False Alarms)</td>
<td>NC-NPC</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>7</td>
<td>3</td>
</tr>
<tr>
<td>Problem Recognition Ratio</td>
<td>P(C</td>
<td>Sf) = NPC/NP</td>
<td>8/9</td>
<td>7/9</td>
<td>9/9</td>
<td>9/9</td>
</tr>
<tr>
<td>Conditional Problem Identification Ratio</td>
<td>P(Sf</td>
<td>C) = NPC/NC</td>
<td>8/9</td>
<td>7/9</td>
<td>9/13</td>
<td>9/16</td>
</tr>
</tbody>
</table>

vi) A component has **not been "covered"** with sufficient number of test cases and it the product of its "Importance" and test-case failure-intensity exceed a threshold. Note that the test-case failure-intensity is adjusted for usage of the component by multiplying it by the (8-level). Seven levels are defined, and it is assumed that "level 1" components will be most frequently used so that the failure intensity observed during non-operational testing will be magnified in operation by factor (8-level).

An example is given in Table 3.1. The table contains columns for individual, as well as joint...
events. The conditions and frequencies would be derived from experimental (historical) information. For example, we see that the "Change Level" and "Importance Level" metrics can recognize 8 and 7 of the 9 problem-prone components, respectively. The "Change Level" has a false alarm rate\(^2\) of 1/9, while "Importance Level" has a false alarm rate of 2/9. The other two individual conditions recognize all problem-prone components, but also exhibit higher false alarm rates. Joint events show improved false alarm rate, as well as better capability to identify problem prone components (NPC/NC). In general, we want the two ratios listed at the end of Table 3.1 to be as close to 1 as possible. We also want the false alarm rate to be as close to zero as possible.

Table 3.2 Example Risk Models

<table>
<thead>
<tr>
<th>Model</th>
<th>Condition</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>if ('test cases per line of code' &lt; 0.08) and ('failed test cases per test case' &gt; 0.04) then 2 else if ('failed test cases per test case' &gt; 0.04) or ('test cases per line of code' &lt; 0.08) then 1 else 0</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>if (((test cases per line of code') &lt; 0.08) and ((8-level)<em>(failed test cases per test case')) &gt; 0.15) then 9/11 else if ((8-level)</em>(failed test cases per test case')) &gt; 0.1 then 9/14 else if ('test cases per line of code') &lt; 0.08 then 9/13 else 0</td>
<td></td>
</tr>
</tbody>
</table>

Any quantitative estimation should be given as an interval rather than as a point estimate (for example, upper and lower 95% confidence bounds).

3.2 Simple Model

We define risk as the probability that an undesirable event takes place and causes an operational loss, multiplied by the magnitude of the loss it causes [Ehr85, Boe89]. We consider the risk given that we know what the current state of software is. Let the undesirable event be the problem-proneness of the software in the field due to one or more categories of faults (for example, the event is transition to state S_f described earlier). The loss, L_f, can be the severity of the resultant class of failures, or any other appropriate measure. Then, given state i, the risk is

\[ R(S_f) = P_i(S_f \mid \ldots) \times L_f \]

If the appropriate information is available, the analysis may be broken down into N failure classes that contribute to the overall problem, that is

\[ R(S_f) = \sum_{j=1}^{N} P_{ij}(S_{fj} \mid \ldots) \times L_{fj} \]

The computed risk value can be used directly, or it can be weighted to reflect some other concerns.

Table 3.2 two summarizes very simple risk models. The score for model 'A' is based on the count of condition events. The model 'B' score is an estimate of the a posteriori probability.

Figures 3.1 and 3.2 show examples of component clustering that might be offered by these risk models. We see that, using the chosen target values, the risk models successfully identify problem-prone products, but also generate some false alarms (items in the rightmost column and below the threshold line).

4. Software Process Control

Advanced software reliability engineering requires active guidance of the testing process based on the quality growth. Many different reliability indicators can be used to establish test stopping criteria, and guide the testing strategy [Dal90, EhP93]. But, many available reliability models are not well suited for evaluation of systems under test with other than their operational profiles. The formulation of software
reliability models, the estimation of their parameters, and the accuracy of their predictions are somewhat controversial issues [Bro92]. Common estimation procedures include maximum likelihood, least-squares and Bayesian approaches [Mus87]. However, the highly correlated nature of the early testing process may require the use of advanced "time-series" analyses to evaluate the reliability growth.

This section briefly examines the role of reliability growth models in the context of multi-phase software process control.

### 4.1 Exposure "Time"

The choice of the "exposure metric" is important. Time is the usual measure. An alternative may be the count of the executed test cases. The underlying assumption is that non-operational testing concentrates on low-level software "operations" which may be better represented by test cases than by the execution time. This metric may be easier to normalize since the number of planned test cases is usually available early in the testing process. The number of unique test cases successfully executed may be combined with the information on the "planned" number of test cases to obtain the test-case "coverage" in terms of the fraction of the planned test cases executed (see the x-axis in Figure 4.2).

### 4.2 Non-Operational Testing

If we assume that early software testing is primarily driven by the desire to cover and verify as many product operations, functionalities, and structures as possible, irrespective of how often they might be used in the field, we can develop "coverage"-driven reliability growth models that operate within the confines of a single testing phase.

The essentials of one such a model, described as a Rayleigh intensity model, are given in [Vou92]. The model has a unimodal intensity profile, and an S-shaped cumulative distribution function. In a more general case, the dynamics of the process translates into a Weibull failure detection model. Weibull-type model, using time as exposure, was considered by Wagoner [Wag73], but not in the context of non-operational profile testing. Also, the Shick-Wolverton model can be interpreted as a special case of the Weibull model class [Shi73, Mus87]. A number of other S-shape models exist [Yam83, Ohb84, Yam86, Toh89].

The unimodal failure-intensity profile, frequently observed during non-operational testing, is in sharp contrast with the monotonously decaying failure-intensity expected from the "classical" reliability models used with operational profile testing. Nevertheless, it is reasonable to expect that, before the testing is stopped, an overall decay in the failure-intensity needs to be observed. Of course, statistical variability of a single sample, that each individual testing effort represents, may mean...
that, in practice, a number of minor modes may be observed in the actual intensity profile.

A growth model can be used to fit and predict failure-intensity during non-operational testing. Figure 4.1 illustrates this. The exposure metric used is the execution time in minutes, and the failure-intensity is in terms of failures per hour. Shown are the observed and calculated (instantaneous) intensity, and the experimental and fitted average failure-intensity. We see that in both cases, and particularly in the latter one, the Weibull model appears to describe the empirical data well.

Figure 4.1  Empirical and modeled intensity profiles obtained during an early testing phase. Exposure is the cumulative test case execution time.

4.3. Process Control

The importance of reliability growth modeling is in the potential use of the predicted parameter, and derived variable, estimates in establishing end-of-phase quality conditions which, in turn, can be used in multi-phase models. To be useful in software process control the feedback, about the potential impact of the current testing (or lack of it) on the operational quality of software, has to be available as early as possible. For example, we would like to be able to answer questions like "What are the estimated failure intensity and residual fault counts for component X at the end of this testing phase?", "Are these values within the expected bounds for this phase?", "What is the impact on the operational reliability of the product?, "How much more testing is required?", etc.

We have seen that the risk models may use test-case intensity, effort, and usage to identify error-prone components. It is interesting to note that a change-level threshold may be an unstable condition, because improvements in the process are intended to destroy the correlation between the size of the change and the problem-proneness. If the parameters required by the risk model, for example the total expected number of failures or the corresponding test-case failure-intensity, can be estimated before a testing phase ends, then it may be possible to use a multi-phase model to assess the impact of the current phase on the quality of software in some future life cycle phase, and direct the effort to components that may need extra testing.

The process involves prediction of end-of-phase quality conditions that describe a software state based on the information before a testing phase is complete. The approach requires modeling of the failure detection process occurring during the non-operational testing, and periodic estimation of the model parameters and derived conditions. The predicted conditions are then used by a risk model to assess the quality of software in some future phase.

Figure 4.2 shows an example of a Rayleigh model fit, made when about 60% of the planned test cases have been executed. Stable estimation bounds may require as much as 50-60% of the test plan to be completed3.

Suppose that the metric of interest is average failure-intensity, and that the condition threshold is 0.3 failures per hour. Figure 4.3, then illustrates a possible process of stabilization of the intensity estimates. It shows the empirical average test-case failure-intensity as a line, and approximate estimate bounds as bars. The estimates are made using the data available up to the point of estimation. We see that as more data becomes available the bounds tend towards the observed average failure-intensity. Inspection of the average failure-intensity graph shows a decreasing trend in the intensity beyond the 80% point.

Note that estimation of parameters may be achieved using least-squares or maximum likelihood. However, inference and computation of the confidence bounds may require more sophisticated techniques, such as time-series analysis.
The final step in the process is to feed the risk model assessment of the components back into the testing process.

Ideally, the reaction to this information would be quick, and correction would be applied already within that testing phase. However, in reality, introduction of an appropriate feedback loop into the software process, and the latency of the reaction, will depend on the accuracy of the feedback models, as well as on the software engineering capabilities of the organization.

For instance, it is unlikely that organizations below the third maturity level on the SEI Capability Maturity Model scale would have processes that could react to the feedback information in less than one software release cycle. Reliable latency of less than one phase, is probably not realistic for organizations below level 4 [Pau93]. This needs to taken into account when the level and the economics of SRE implementation is considered.

Acknowledgments

The first author is grateful to Dr. W. Jones for many invaluable discussions of the software reliability engineering issues.

About the Authors

Mladen A. Vouk received B.Sc. and Ph.D. degrees from the University of London (UK). He has extensive experience in both commercial software production, and academic computing environments. He is currently an Associate Professor of Computer Science at North Carolina State University. His research and development interests include: software quality assurance and risk management, software testing and reliability, software process modeling, software fault-tolerance, and scientific software issues. He regularly teaches courses in software engineering, software testing and reliability, software risk management and programming language C. Dr. Vouk is the current secretary of the IFIP Working Group 2.5 on Numerical Software. He is a senior member of IEEE, and a member of IEEE Reliability, Computer and Communications Societies, IEEE TC on Software Engineering, ACM, ASQC, and Sigma Xi. He is an associate editor of IEEE Transactions on Reliability. E-mail: vouk@adm.csc.ncsu.edu, Tel: 1-919-515-7886.

Kuo-Chung (K. C.) Tai received his B.S. degree in Electrical Engineering from National Taiwan University in 1970 and his Ph.D. degree in Computer Science from Cornell University in 1977. He is currently a Full Professor of Computer Science at North Carolina State University. He has been with NCSU since 1976 except that from 1989 to 1991, he served as the director of Software Engineering Program at the National Science Foundation in Washington.
DC. His currently research interests include software testing, concurrent programming languages, and analysis, testing, and debugging of concurrent software and communication protocols. He is an associate editor of Computer Languages, International Journal of Software Engineering and Knowledge Engineering, and International Journal of Computer and Software Engineering. He is a co-chair of 1994 International Conference on Parallel Processing. His Internet address is kct@csc.ncsu.edu and his phone number is (919) 515-7146.

References


