ABSTRACT
The paper presents some new contributions to software reliability models. Terms and definitions for software reliability and associated concepts are proposed. The authors discuss the possibilities of taking special properties of an SPC switching system into account. Based on this discussion they propose mathematical models, evaluate contributing factors and derive suitable characteristics.

The model validation is treated in connection with a case study.

1. INTRODUCTION

1.1 GENERAL

Experience has taught us that the software reliability and a correct handling of the system contributes to the overall SPC system reliability at least as much as the hardware reliability. Terminology and techniques for qualitative and quantitative analysis of reliability and associated characteristics of hardware are well developed, whereas similar aspects of software and handling still need further development.

One obvious use of analytical methods is to quantify these characteristics of an SPC system to guide system designers in their search for good designs.

An equally important application area for analytical methods is the development of tools for quality assurance. Tools for prediction and extrapolation for the operational phase are also needed.

To allow an adequate description of reliability and associated aspects of an SPC telephone exchange software, a definition of fundamental concepts is required in order to position the concept set and to that set allocate a suitable flora of terms. This base might then also facilitate a unification of modelling principles of software and hardware.

The term reliability usually means different things to different people. Therefore, before we are able to talk about the software reliability we first have to define these concepts.

1.2 SOFTWARE

As to software, this term is by different authors referring to just the program or to program and data combined. We distinguish two main types of software, program software and data software.

Program software nowadays shows a hierarchical and modular structure: a monitor program supervises a number of more or less standardized program modules (macros or subroutines). This hierarchy is also found in the design process (top-down or bottom-up structured programming). You may for instance talk about a program at different levels of implementation:

1. The algorithm
2. The (very) program (code)
3. The object coded program

In a processor system, data software is characteristically of rather different types in the following sense:

1. Parameter settings in standardized program modules
2. Environment defining data
3. Environment generated data

The distinction between these types may be rather diffuse, but the classification mainly is to show the frequency by which data change.

Many suggestions have been made in order to improve the software reliability and this has also, despite the increasing complexity of the computer systems, resulted in more trouble-free systems. However, we still lack of generally accepted procedures for a more detailed analysis of failure data of the sort you find for instance in the classical (hardware) reliability theory.

1.3 SOFTWARE RELIABILITY

The definition of reliability as a qualitative characteristic is: The ability of an item to perform its required function under stated conditions for a stated period of time [3]. The term is also, however, somewhat incorrectly and imprecisely used to denote a wider area, including properties such as availability and maintainability. Besides, it is also used as a measure i.e a quantification of reliability.

The maintainability of an item is its ability, under stated conditions of use, to be retained in or restored to a state in which it can perform its required function, when maintenance is performed under stated conditions and using prescribed procedures and resources [4]. The maintainability support is the ability of the maintenance organization to mobilize resources to perform maintenance required by the item [5].

The trafficability of an item is its ability - under stated use conditions - to fulfill operational traffic demands [5], [9]. The availability is the ability of an item - under combined aspects of its reliability, maintainability and maintenance support - to perform its required function at a stated instant of time or over a stated period of time [4].

The effectiveness of an item is its ability - under combined aspects or its availability and trafficability - under stated use conditions to fulfill operational traffic demands [5], [9].

In order to take into account also the man-machine interactions the following concepts may be used: Operability, the ability of an item to be successfully operated when operation is performed under stated conditions and using prescribed procedures [9], Operator performance, the ability of the operator to successfully operate the item in accordance with prescribed procedures [9].

Of the above-mentioned concepts, only reliability has been standardized by IEC up to now.
When the terminology is applied to SPC technology it may be possible to subdivide reliability, maintainability and hardware reliability into two separate parts, software and software. You can then talk about both software reliability and hardware reliability.

The concepts software reliability, software maintainability and software maintenance support seem to be possible to define along the same lines as above. Mainly, only the meaning of the term "item" needs to be altered. Accordingly the following terms and definitions are attempted:

Software reliability - the ability of a software item to perform its required function under stated conditions for a stated period of time. Comment: Stated conditions refer to the conditions under which reliability of the use of the software as determined by data environment. The term "software item" denotes any piece of software that can be separately considered.

Software maintainability - the ability of a software item, under stated conditions of use, to be retained in or restored to a state in which it can perform its required function, when maintenance is performed under stated conditions and using prescribed procedures and resources. Comment: This includes the ability of the software to detect and give notice when irregularities are present and the ease with which temporary and definite changes of the software can be made.

Software maintenance support - the ability of the maintenance organization to mobilize resources to perform maintenance required by the software item. Comment: These are the properties of the organization for software change activities, such as personnel and test facilities.

1.4 SOFTWARE ERRORS AND FAILURES

With hardware it is rather easy to decide upon what is a component and whether a certain component is defective or not. Also, when the system fails, i.e., a system failure occurs, it is in general possible to trace the cause of this failure to a certain defective such component. One main reason for this is that the physical (hardware) structure very well agrees with the functional structure of the system. As regards software, the situation is much more complicated. For instance, interactions between programs and/or data modules must be taken into consideration.

A revision of the currently used terminology seems to be justified. In this paper, however, we use the following definitions: Error - any nonconformance of the software item with specified requirements. Failure - the event during software testing and operational use when the existence of an error is detected (error detection).

There is a fundamental difference between hardware and software failures. Hardware failures are due either to design errors (components or parts wrongly used) or to causes inherent in the component (sudden failures or degradation failures). Software failures are only caused by software defects (errors) introduced during design and production (including software changes) or by external influences on the software during operation.

Many articles on quantitative description of software reliability deal with the problem of estimating the number of errors in a program. In this article we present another quite different approach that may be characterized as a macroscopic point of view as opposed to the microscopic "find-the-guilty-and-count-him" approach. The starting point is a record of failures made up as a part of the operating routine (or during the test phase). The description of the reliability then in the first place is the frequency of these failures or the failure rate. In the second place, it is a classification of these failures into different causes and modes, such as the ones mentioned above. Thirdly, we may want a description that overcomes our inability to a detailed description of the failure process, i.e., that takes the "randomness" into account.

While we have the SPC program software in mind, the models and methods we present should apply to different kinds of failures (according to the above classification for instance).

2. MODELS

2.1 FAILURE LIABILITY

A specific property of program software is that it operates upon input data (signals) and transforms this input into output data (signals) in such a way that different output data (signals) necessarily come from different input data (signals). By contrast this is not the case for hardware.

In other words, if X denotes the set of possible input data and Y denotes the set of possible output data, then a program software (sub) system can be described as a function P: X → Y. (Here 'possible' includes erroneous in both cases). When executed the program transforms the actual input into an output y = P(x). We shall assume that the program P is recurrently executed and that the set X is so defined that at each execution, one and only one x ∈ X is presented to P.

If a program P does not do what we want it to do, then in the first place we have a specification (or at least a desire, an intention) P* for P and in the second place we have inputs x ∈ X for which P(x) ≠ P*(x).

Here P and P* are supposed to be defined on the same X and ranging into Y. This may in practice cause some ambiguity in the definition of X and Y. However, since we are interested only in the correctness of P, this difficulty can be overcome by taking X and Y sufficiently large.

As a first step towards a quantitative measure of the reliability of the program P we define the failure generating inputs, F:

\[ F = \{ x \in X : P(x) \neq P^*(x) \} \]

For any given pair (P, P*) the failure generating input set is well-defined and easily determined in principle, if not always in practice.

The simplest way to get a numerical measure of the "size" of F is perhaps to calculate the proportion of x's belonging to F,

\[ f = \frac{|F|}{|X|} \]

where |F| and |X| are the numbers of elements in these sets. We shall call this proportion the failure liability. The concept may be extended to a subset of X, X_i ∈ X:

\[ f_i = \frac{|F \cap X_i|}{|X_i|} \]

which we call the conditional failure liability on X_i.

It is now possible to give the failure liability an operational meaning, by taking into account the frequencies of the various inputs. This may be done to any desired extent by dividing the input set X into disjoint subsets X_1, ..., X_n so that X = U X_i, and by introducing the probabilities

\[ p_i = P_r(x \in X_i) \quad (i = 1 \ldots n) \]

calculated in the sense of which x that is present upon an distribution of P. Since we have assumed exactly one x ∈ X to be present, we have \[ \sum p_i = 1 \]. We now define the input frequency consistent failure liability,

\[ f = \sum_{i=1}^{n} f_i \cdot p_i \]

where f_i is the conditional failure liability on X_i (i = 1 \ldots n)
We are aware of various practical difficulties that will be encountered when a specific program shall be assigned a failure liability. It is hard to see, however, why a measure of the reliability of a program should not primarily depend on $P$, $P^m$ and on the input data distribution, and we consider the concept of failure liability a basic characteristic for the determination of software reliability.

The terminology is our own, but ideas in the same direction are found in [7].

2.2 FAILURE RATE

By failure in program software we mean the occurrence of an input-output pair $(x, y)$ such that $y \neq P^m(x)$ that is we do not obtain what we want to obtain. By definition $y = P(x)$ and so we have equivalently an occurrence of an input $x \in \mathbb{F}$. Of course the presence of an $x \in \mathbb{F}$ and the execution of $P$ are necessary for a failure of $P$.

By failure rate we here mean the average number of failures per unit time (during an interval of time).

If the program $P$ is executed $c$ times per unit time and the (input frequency consistent) failure liability is $\bar{f}$, then the failure rate $v$ becomes:

$$v = c \cdot \bar{f}$$

For a program software system with non-overlapping subprograms $P_1 \ldots P_N$ the system failure rate $v_s$ becomes

$$v_s = \sum_{i=1}^{N} c_i \cdot \bar{f}_i$$

where $c_i$ and $\bar{f}_i$ are the execution rates and failure liabilities for $P_i$ (cf ref [7]).

These trivial relations indicate a method to evaluate system reliability from subsystem parameters and stress the importance of the failure liability concept.

2.3 FAILURE DATA ANALYSIS

So far we have factored the failures into these two obvious provisions, an execution of the program and an erroneous input-output pair $(x, y)$. We may, however, be interested in the underlying reasons for $P$ not being $P^m$. A number of such reasons may be listed, for instance,

1. $P^m$ was equal to $P$ when $P$ was implemented but has now changed. (In fact $P^m$ is in practice never completely specified due to human limitations).
2. $P$ was equal to $P^m$ initially but has been changed.
3. $P$ is equal to $P^m$ on the algorithm level, but the programmer did not implement it correctly.
4. $P$ is equal to $P^m$ on the source program level, but was erroneously translated into object program.

By tracing such inherent causes to their origin we may detect weak links and take due action.

Another reason for looking deeper into inherent and direct causes is that certain subprograms may be similar though distinct. One may expect such similar subprograms to have similar reliability properties. On the other hand, it is known that different programming techniques and different programmers cause rather different amount of troubles. Also the complexity and size of a subprogram are of vital influence on the failure liability.

If we can model these similarities and dependences, then we may be able to predict reliability performance for systems in the design phase.

To make such an analysis we classify (sub)programs into a number of disjoint classes $A_1 \ldots A_n$, representing different levels of a potential influent factor "A", number of instructions for instance. We then want to decide whether the factor has a significant influence on the reliability and, if this is the case, to describe the influence quantitatively.

If the analysis is to be based on failure data it is necessary that each failure can be allocated to a specific (class of)subprogram(s). This may be hard to fulfill, if failure data is collected without special attention to this condition. A careful division of the program system into subsystems may facilitate this allocation. Of course there will be failures that do not depend on a specific subprogram. Such failures must then be identified and ruled out from a subprogram analysis.

Apparently any kind of influence will be disturbed by the randomness of the failures. We therefore have to rely on statistical techniques for the evaluation of observed data. Such techniques are based on a probability model of the phenomenon under study. The choice of such a model is always a compromise between adequacy (fit) and tractability.

2.4 THE EFFECT OF ERROR-CORRECTION

In practice the failure of a program $P$ will cause an error-tracing-correcting procedure to be undertaken, eventually resulting in a change of $P$, hopefully towards $P^m$. Therefore software failure data only exceptionally comes from a fixed program $P$, but rather from a sequence of programs $P_1, P_2 \ldots$. The items of the sequence will be rather similar though.

The effect of this error-correction is marked in the beginning of a program's life. As an example consider the failure data records of figure 1 that are typical and reveal a roughly exponential decrease of the failure rate.

It is tempting to treat the initial number of errors, $N$, as an unknown parameter and try to estimate $N$ from observations by some standard estimation procedure. This problem has been studied by [3, 5].

Unfortunately "the number of initial errors" is a very ambiguous quantity. It is initially not possible, not even in principle, to delimit the "errors", less possible to number them. Remains to treat $N$ as a non-interpretable parameter.

2.5 A MODEL FOR DECREASING FAILURE RATE

In this section we consider failures produced by a single (sub)program $P$ with failure liability $\bar{f}$ and (known) execution rate $c$.

To avoid complications due to execution rate dependence on traffic variations, we take a sufficiently long unit interval of time, preferably one week, to make the execution rates stationary, and record the number of failures in successive unit intervals of time (cf fig 1).

As errors are corrected, the failure generating input set $F$ shrinks. Therefore we allow the failure liability to depend on time and put

$$\bar{f}_t = \text{failure liability at time } t \ (t=1,2,\ldots)$$

The failure rate $\nu$ consequently depends on time,

$$\nu = \nu_t = c \cdot \bar{f}_t \ \ t = 1,2,\ldots$$

Put $n_t$ = number of failures in time interval $t \ (t=1,2,\ldots)$. 

ITC8 412-3
A general model for describing the sequence $n_t$ that takes random fluctuations into account, is obtained by considering a fixed epoch $T$ (i.e., the unit intervals $t = 1 \ldots T$) and assuming the $n_t$'s to be independent Poisson distributed with parameters $\lambda t$ ($t = 1 \ldots T$). In other words the joint probability frequency function

$$P(n_1, \ldots, n_T) = \prod_{t=1}^{T} \frac{e^{-\lambda t} (\lambda t)^{n_t}}{n_t!}$$

By specializing on the $\lambda t$'s we can model different kinds of time dependence. The most special (and simplest) case is to put all $\lambda t$ equal:

$$\lambda t = \lambda \quad i = 1, \ldots, T$$

From this familiar case we generalize by putting $\lambda t = \mu t \cdot \theta^t$ and $\mu t = \mu \cdot \theta^t$ ($\mu > 0$, $\theta > 0$), thus obtaining

$$\lambda t = \mu \cdot \theta^t \quad (t = 1, \ldots, T)$$

and get

$$P(n_1, \ldots, n_T) = \prod_{t=1}^{T} \frac{e^{-\lambda t} (\lambda t)^{n_t}}{n_t!}$$

where

$$P_t(\theta) = \frac{(1 - \theta)^{T-1}}{1 - \theta^T}$$

and

$$\lambda(\theta) = \mu \cdot \theta(1 - \theta^t) \quad (\theta > 0)$$

Here $\theta(1 - \theta^t)$ is to be interpreted as $T$ when $\theta = 1$.

Note that $\theta = 1$ corresponds to the constant failure rate case.

This joint probability factors into a Poisson distribution for the total number of failures $n = \sum n_t$ and a multinomial distribution with geometrically decreasing ($\theta < 1$) or increasing ($\theta > 1$) probabilities for the distribution of these failures among the unit intervals.

From this observation we immediately conclude that the expected number of failures in the $t$'th interval conditionally on the total number of observations $n$ is

$$E[n_t | n] = n \cdot \frac{1 - \theta^t}{1 - \theta^T}$$

$t = 1, \ldots, T$

Thus for $\theta \leq 1$ we have a model that accounts for an (exponential) decrease in failure rate and contains the constant failure rate case ($\theta = 1$).

The constant failure rate model is a natural first approximation, and in this spirit the generalization is a natural second order approximation for the type of data considered.

### 2.6 A TIME-SPACE MODEL

Typically an SPC exchange program software consists of subprograms with different levels of standardization. Some subprograms are almost identical in different exchanges and therefore thoroughly tested. Such programs will cause very little trouble already from the outset. Other subprograms are tailored to special demands on the specific exchange and therefore unique. These programs may cause trouble, not only during the test phase. In between these extreme cases there are intermediate cases of course.

It is anticipated that when subprograms are classified in the above respect, the number of failures per unit of time and instruction becomes a rather significant figure that differs between classes but is stable within classes.

To model these effects, we suppose that subprograms are uniquely classified into $K$ categories. $A_1 \ldots A_K$ of standardization. We consider only failures that can be uniquely traced (allocated) to a subprogram. The size in terms of number of instructions ($I$) and the execution rate ($C$) of each subprogram are supposed to be known.

We start from observations during $T$ unit intervals of time ($e.g.$ weeks). Put

$$n(t, P) = \text{no of failures in time interval } t$$

caused by (traced to) subprogram $P$ ($t = 1 \ldots T$ and $P$ ranges over the subprograms).

As before we assume the $n(t, P)$'s to be independent and Poisson distributed with parameters $\lambda(t, P)$ respectively. It becomes necessary to specialize rather drastically on these parameters, since the amount of data is fairly limited. Therefore we put

$$\nu(t, P) = C_P \cdot \theta^t \cdot \lambda_k \cdot \mu_k$$

where

$$P = \text{program identifier}$$

$C_P$ = execution rate of program $P$

$L_P$ = no of instructions of program $P$

$k = \text{class of program } P$

$t = \text{(unit interval of) time}$

and $\theta_1, \ldots, \theta_K, \lambda_1, \ldots, \lambda_K, \mu_1, \ldots, \mu_K$ are unknown positive parameters. Thus we have $3K$ parameters.

The introduction of the factor $C_P$ is explained in section 2.2, and the factors $\theta^t$ are motivated by the discussion in section 2.5. The factors $\lambda_k$ are motivated by similar ideas. Here we expect $\lambda_k > 1$ though. Less obvious is the simple multiplication of these factors which is more or less a force majore to keep the number of parameters within reasonable limits.

The joint probability of $\{n(t,P), t = 1 \ldots T, \forall P\}$ becomes under these assumptions

$$\prod_{t=1}^{T} \prod_{P} \frac{(C_P \cdot \theta^t \cdot \lambda_k \cdot \mu_k)^{n(t,P)}}{n(t,P)!} \cdot \exp(-C_P \theta^t \cdot \lambda_k \cdot \mu_k)$$

The model can be simplified further if the error-correcting rate parameters $\theta_P$ and/or the program length dependence parameters $\lambda_k$ can be considered equal. The number of observations required to test this will be in the order of $300 \cdot K \cdot e$ with $K = 3$ some thousand failures. From example we see that while the proposed model comprises many simplifications, it is out of question to consider a more general model. At least we have a model that, perhaps in a rough way, reflects the time dependence, the complexity dependence and the test status dependence. We think these are the major factors that affect the reliability of a program.
3. DATA ANALYSIS AND EXTRAPOLATION

We conclude this paper with an illustrative case study to indicate how the above models may be used for analysis and extrapolation of software failure data.

3.1 THE DATA

The failure data in the study were collected during the first 12 months of operation of the AKE 13 exchange at Fredhäll, Stockholm [26]. Out of 82 reported failures 55 failures were considered program dependent. (No regard has been paid to the effect of a failure, e.g. whether or not the traffic handling was disturbed).

The AKE program was divided into the two main parts, APT (traffic handling, telephony operation and maintenance programs) and APZ (operating systems programs) and the failures were traced to these parts. The resulting failure data is displayed in fig 1.

It should be understood that the amount of data in this case study in fact is too small for a serious statistical analysis. Our purpose is merely to indicate the possibilities of the proposed models and methods.

3.3 TESTING FOR FAILURE RATE DECREASE/INCREASE

At a constant failure rate the accumulated numbers of failures extend along a straight line from (0,0) to (T,n) in an accumulated-frequency-diagram. Here T is as before the observation period and n = the total number of failures observed.

If the accumulated numbers fall markedly above (below) this line it is an indication of decrease (increase) in the failure rate. To test statistically if such a departure from uniformness is significant, compute

\[ \bar{m} = \frac{1}{n} \sum_{i=1}^{T} t \cdot n_t \]

Then \( \bar{m} \) is the mean of the time to failure distribution \( (n_1 \ldots n_T) \).

If we assume that the \( n_t's \) come from a uniform distribution of the \( n \) failures among the \( T \) intervals of time, the expectation and standard deviation of \( \bar{m} \) (considered a random variable) are

\[ E(\bar{m}) = \frac{T+1}{2} \]

\[ \sigma(\bar{m}) = \frac{T}{\sqrt{12} \cdot n} \]

Further it may be shown that \( \bar{m} \) is approximately normally distributed (\( \sigma > 20 \)) under the hypothesis.

Thus we reject the model of constant failure rate in favour of a significant increase/decrease (at 5% significance level) if

\[ |\bar{m} - \frac{T+1}{2}| > \frac{T}{\sqrt{3}n} \]

This gives for the AKE data (\( T = 12 \) months):

<table>
<thead>
<tr>
<th>n</th>
<th>( m )</th>
<th>( m - \frac{T+1}{2} )</th>
<th>Significant</th>
</tr>
</thead>
<tbody>
<tr>
<td>APZ</td>
<td>16</td>
<td>5.375</td>
<td>-1.125</td>
</tr>
<tr>
<td>APT</td>
<td>39</td>
<td>3.535</td>
<td>-3.167</td>
</tr>
<tr>
<td>APZ+APT</td>
<td>55</td>
<td>3.927</td>
<td>-2.573</td>
</tr>
</tbody>
</table>

Fig 1 Observed and expected (according to the model) failure frequencies. Intervals = 4 weeks
3.4 ESTIMATION OF θ AND CHECK FOR EXPONENTIAL DECREASE (INCREASE) IN FAILURE RATE

When checking the assumption of an exponential decrease (increase) in the failure rate, we must confine ourselves to looking for severe violations of this model. If there seem to be such violations, then we may try to divide the observation period into subperiods in such a way that the model applies to these subperiods separately. Should this not work, then there is presumably no statistical evidence in the observations and we are forced to give up statistical treatment.

The maximum likelihood (and moment as well) estimators of ν and θ in the model

\[ P(n_{1},...,n_{T}) = \prod_{t} \left( \frac{(v \cdot \theta^{t})^{n_{t}}}{n_{t}!} \cdot e^{-v \cdot \theta^{t}} \right) \]

are obtained as solution of the equations

\[ n = \sum_{t} n_{t} = \nu \cdot \frac{(1 - \theta^{T})}{1 - \theta} \]
\[ m = \frac{1}{n} \sum_{t} t \cdot n_{t} = \frac{1}{1 - \theta} - \frac{T \cdot \theta^{T}}{1 - \theta^{T}} \]

These equations are valid for all \( \theta > 0 \) if the right hand expression is interpreted as \( (T + 1)/2 \) for the special case \( \theta = 1 \) (which should be treated separately, but is obtained as a limiting case for \( \theta \to 1 \)).

When θ has been estimated, we can estimate the expected frequencies \( \hat{n}_{t} = E(n_{t}) \) by

\[ \hat{n}_{t} = n \cdot \frac{(1 - \hat{\theta})^{t-1}}{1 - \hat{\theta}} \quad (t = 1, ..., T) \]

where \( \hat{\theta} \) is the maximum likelihood estimate for θ, and compare these with the corresponding \( n_{t} \)'s.

As an aid in this comparison we may also calculate the \( X^2 \)-criterion

\[ X^2 = \sum_{t} \frac{(n_{t} - \hat{n}_{t})^{2}}{\hat{n}_{t}} \]

and find the significance from a \( X^2 \)-distribution with \( T - 2 \) degrees of freedom.

When applied to the AKE data these methods yield

<table>
<thead>
<tr>
<th>ν</th>
<th>θ</th>
<th>X^2</th>
<th>df</th>
<th>P-value</th>
</tr>
</thead>
<tbody>
<tr>
<td>APZ</td>
<td>2.36</td>
<td>.9080</td>
<td>1</td>
<td>&gt; 0.90</td>
</tr>
<tr>
<td>APT</td>
<td>15.47</td>
<td>.7199</td>
<td>4</td>
<td>&gt; 0.05</td>
</tr>
<tr>
<td>APT+APT</td>
<td>16.16</td>
<td>.7822</td>
<td>4</td>
<td>&lt; 0.01</td>
</tr>
</tbody>
</table>

Comments: Due to the small number of observations, intervals had to be pooled before the application of the \( X^2 \)-test. Hence the loss of degrees of freedom (d.f.). - The fit to the complete unstratified set of data is unacceptable. The stratification and allowance for separate ν's and θ's yield satisfactory fit.

3.5 EXTRAPOLATION

Suppose we have observed the failure process of a program P during \( T_{1} \) intervals of time and want to give it a figure of merit. If the multiplicative Poisson model discussed above is adequate we can estimate the initial failure liability \( \nu \) and the intrinsic failure (liability) decrease rate parameter \( \theta \) as indicated above and use these.

However, it is often desirable to get a figure that more directly reveals how the system will perform in the nearest future, say the number of failures that may be expected during the following 2-4 intervals of time.

Such an extrapolation of data always makes an assumption about things to 'continue behave in like manner'. In our case this assumption is in effect that we assume our basic model, which in an inductive way depends on the epoch \( T \), to be valid for any epoch \( T \).

Thus let the total epoch \( T = T_{1} + T_{2} \) be divided in an observational interval \((T_{1}, T_{2})\) of length \( T_{1} \) and an extrapolational interval \((T_{1} + T_{2}, T + T_{2})\) of length \( T_{2} \).

Let further

\[ n(t) = \text{no of failures in time interval } t \]
\[ n_{1} = \sum_{t} n(t) \]
\[ m_{1} = \sum_{t} t \cdot n(t) \]
\[ n = \sum_{t} n(t) \]
\[ m = \sum_{t} t \cdot n(t) \]

Thus \( m = m_{1} + m_{2} = T_{1} \cdot n_{2} \)

Applied to the total epoch \( T \) our model becomes

\[ p(n(t), t = 1, ..., T) = \prod_{t} \left( \frac{(v \cdot \theta^{t})^{n(t)}}{n(t)!} \cdot e^{-v \cdot \theta^{t}} \right) \]

The marginal distribution for \( n_{2} \) turns out to be a Poisson distribution with parameter (= expectation)

\[ E(n_{2}) = \nu \cdot \frac{(\theta^{T} \cdot \theta^{T})}{1 - \theta} \]

The unknown parameters may be estimated from the observed values on \( n_{1} \) and \( m_{1} \) by the method given above. If these estimates are inserted into the expression for \( E(n_{2}) \), we get a reasonable extrapolation:

\[ n_{2} = n_{1} \cdot \frac{\hat{\theta}^{T} \cdot \hat{\theta}^{T}}{1 - \hat{\theta}^{T}} \]
For illustrational purpose we simulated a 1-interval-ahead extrapolation of the failure frequency with the AKE data. For the APT part we used the method above, but for the APZ part we estimated by the observed over-all rate, \( u(T_i)/T_i \), since this sequence was considered stationary (above). When these extrapolations were added we obtained:

<table>
<thead>
<tr>
<th>Time</th>
<th>5</th>
<th>7</th>
<th>9</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>Extrap</td>
<td>2.83</td>
<td>1.97</td>
<td>1.74</td>
<td>1.68</td>
</tr>
<tr>
<td>Actual</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>2</td>
</tr>
</tbody>
</table>

Allowance should be made for deviations of order 2-3 times the square root of the extrapolation. You may also compare with the naive extrapolations, i.e., extrapolations based only on the overall failure rates, that here become 8.20, 6.14, 5.44 and 4.91 respectively.

4. CONCLUSION

We have presented models and methods for the description, analysis and extrapolation of software reliability as measured by failure rates. The models include a system parameter, the failure liability that may be taken as a figure of merit with respect to the reliability and used at least for comparative purposes. The methods rely on customary statistical praxis and may be developed further for more detailed analysis.

It should be understood, however, that data on software failures very often is scarce for refined models and detailed analysis. Therefore our models and methods (as well as any other model and method) only can serve as rough approximations to the state of things.

REFERENCES


[4] IEC TC 56 (IEV) (Secretariat) 92: Third set of additional terms for inclusion in publication 271 (second edition)

[5] IEC TC 56 WG1 (Strandberg) 1: Proposal: Broad outlines for a revision and rearrangement of terms and definitions


