STATISTICAL MODELLING OF THE RELIABILITY OF SPC SOFTWARE

by T. Downs

Department of Electrical Engineering
University of Queensland
St. Lucia, Qld. 4067, Australia

ABSTRACT

A major objective of this paper is to draw attention to some of the issues surrounding the software reliability problem and their implications regarding the analysis of teletraffic systems which include SPC. A new approach to software reliability modelling, which allows assessment of the effects of differing testing strategies, is described. A method of approximately upper-bounding the step increase which occurs in transferring software from the test phase to the operational phase is also presented. In addition, we address the important question of whether or not exponential distributions can be used to accurately model the intervals between failure times for software in the operational phase. This question has considerable significance in relation to computation of end-to-end connection probabilities in SPC networks.

1. INTRODUCTION

Traditional network dimensioning procedures are aimed at ensuring that specified blocking criteria are satisfied on final choice routes during normal working conditions. As pointed out in [1], recent trends in system design may render this approach somewhat less than satisfactory. Increases in traffic demands are resulting in larger, more efficient networks which, through the fact that they operate at high occupancy levels, cause the network to become more vulnerable to equipment outages. A further contributing factor to this vulnerability is the trend toward provision of trunk groups in large modules.

In order to account for these developments, the author of [1] recommends that dimensioning be carried out in terms of end-to-end connection probability, a measure which incorporates the effects of congestion due to equipment failures. Other authors (see [2]) recommend the use of other system performance measures (but again incorporating equipment reliability) as a basis for network design and analysis.

If the dimensioning of teletraffic networks is, in the future, to take account of equipment reliability, then, given the current proliferation of SPC systems, it must also take into account the reliability of SPC software. It is the purpose of this paper to discuss the problem of statistical modelling of software reliability with particular reference to SPC software.

In what follows the term "software failure" implies an unacceptable departure from requirements of a program under test or during operation. It is assumed that software failures are caused by programming errors and such errors are referred to as "faults" throughout the paper. Except where otherwise stated, the time variable used in the models developed is computer run time.

2. EXISTING MODELS AND THEIR IMPLICATIONS FOR TELETRAFFIC SYSTEMS

Most of the models which have so far appeared in the literature assume that when a software failure occurs, some effort is made to rectify the fault before the system is restarted. Thus, if the time variable used is computer run time, the software failure process constitutes a point process in time with the software being (instantaneously) subjected to some modification at each failure point.

The first such model to appear in the literature was described by Jelinski and Moranda [3] (we will henceforth refer to it as the JM model) and this model made the simplifying assumption that each time a failure occurred the fault causing that failure was located and rectified before the system was restarted. As we shall see later, this assumption can be relaxed fairly readily.

Another assumption underlying the JM model, which has considerable significance in relation to the analysis of teletraffic systems, relates to the instantaneous failure rate function. This function, usually denoted by $\lambda(t)$, may be defined as

$$\lambda(t) = \lim_{\Delta t \to 0} \frac{\Pr \{ \text{A system of age t will fail in interval (t,t+\Delta t) given that it survives to time t} \}}{\Delta t}$$

The authors of [3] use this function to represent the reliability of a software system during each failure-free interval. Thus, they take, as time origin for equation (1), the time of last failure. This approach is somewhat at variance with recently agreed definitions [4] but since practically all workers in the software reliability field use the failure rate function in this way, we will, for ease of discussion, adopt the same approach.

The assumption made by Jelinski and Moranda which has significance for teletraffic analysis is that the failure rate of a software system at any
given time is proportional to the number of faults in the software system at that time. This assumption implies that the failure rate is constant during the intervals between failures (because the number of faults in the system does not change during these intervals). It immediately follows that the assumption implies that the times between failures of the software system are exponentially distributed.

Jelinski and Moranda did not provide a very strong justification for their assumption and it has recently come in for some quite sharp criticism from Littlewood in [5]. Before we look at some of the criticism, it is worth noting that several other authors have adopted essentially the same assumption [5-8] and that a model of this type has been shown to give good results [9].

From a teletraffic system point of view, it is very important to study more closely the assumption that times between software failures are exponentially distributed. If it can be demonstrated beyond reasonable doubt that the assumption is satisfactory then the performance analysis of teletraffic systems including SPC should be quite tractable. This is because both the traffic process and the failure times of system hardware can normally be assumed to be governed by exponential distributions. Thus, if software failure times are also governed by exponential distributions, a teletraffic system including SPC can be modelled as a simple Markov process. The number of system states may, of course, prove something of a problem, but that is not our concern here.

The major criticism made by Littlewood of models of the JM-type is that the assumption that software failure rate is proportional to the number of faults, implies that each fault has an equal chance of being located in any given time interval and hence that the software is executed in a perfectly uniform fashion over time. SPC software is certainly not uniformly executed; for instance, the software associated with call-handling is much more heavily executed than the maintenance software. Thus, if we stick to the definition of software failure given in section 1, we cannot expect the failure rate of SPC software to be proportional to the number of inherent faults. However, there are two points which we must bear in mind. First of all, not all software faults are going to affect the traffic through an SPC exchange, and this point will be considered in more detail later. Secondly, the fact that the failure rate is not proportional to the number of inherent faults need not necessarily imply that it does not remain constant, or nearly so, during the intervals between failures.

As regards the second point above, we have to concede that the failure rate of software which is not uniformly executed is unlikely to remain constant in the intervals between failures. In the test phase of SPC software, the code responsible for call-handling is usually much more rigorously tested than most of the maintenance software. Thus, in the operational phase (the phase of interest to teletraffic engineers) those sections of software which have been thoroughly tested will be executed most of the time (and the failure rate should be relatively small) but occasionally sections of code which have not been heavily tested will be executed (and the failure rate may become significantly higher). Littlewood [10] has recently attempted to model this situation by making the assumption that each fault makes a random contribution to the failure rate. By making various assumptions which are largely for mathematical tractability, Littlewood arrives at a model which indicates that the time between software failures has a Pareto distribution.

The Pareto and exponential density functions are very similar in shape, the major difference being that the Pareto density function has rather more probability mass in its tail. Littlewood's model constitutes a first attempt at modelling the phenomenon of non-uniform execution. The model is not really suitable for the calculation of end-to-end connection probability because it does not take account of the fact that many faults in the lightly-used area have no effect on traffic. Later in this paper, a rather different approach to this problem will be described.

3. OBJECTIVES OF THIS PAPER

A major objective of this paper is to draw attention to some of the issues surrounding the software reliability problem and their implications regarding the analysis of teletraffic systems which include SPC.

In the previous section it was pointed out that Jelinski and Moranda [3] developed a model based upon the implicit assumption that the intervals between failures were (independently*) exponentially distributed. Several other models were subsequently produced which were based upon essentially the same assumption [6-8]. No clearly-defined premises were advanced to support this assumption. In this paper, models will be developed whose underlying assumptions are more clearly defined; in addition, these models will allow a more realistic assessment to be made of the effects of testing and debugging upon the performance of SPC software when handling traffic. Previous models have been assumed equally-applicable to both the test and operational phases even though it is surely true that the statistics of the failure process will differ in the two phases.

In the following sections the problem of modelling the distribution of faults over the software is first discussed and an elementary argument leads to a binomial distribution of faults over the logic paths through the software. The manner in which this distribution is modified by different testing (and debugging) strategies can then be studied; three such testing strategies are considered here. This kind of study allows the prediction of software performance (in the reliability sense) after testing is completed and the software is employed in traffic handling. One difficult aspect of this prediction relates to the fact that when SPC software is transferred from the test phase to the operational phase, this transfer is invariably accompanied by a (usually large) step increase in software failure rate. Application of traditional statistical techniques to failure-time data collected during the test phase is of no assistance in predicting the size of the step.

* Though not mentioned in section 2, independence of the failure time intervals is also assumed by Jelinski and Moranda.
increase (traditional methods simply predict a continuation of the downward trend in failure rate experienced during the test phase). A method of determining an approximate upper bound on the step increase is described in section 6. In section 7 it is pointed out that although data collected on software pertaining to early SPC systems appears to indicate that failure times are approximately exponentially distributed, the later systems, which have been developed using the more recent ideas of structured design [11] may exhibit different characteristics. There does not appear to be very much software reliability models have paid little or no attention to the actual distribution of faults over the software. As a consequence, these approaches have essentially involved the proposal of a distributional form for the (random) lengths of intervals between failures, and this form has been assumed to hold throughout the lifetime of the software. In this paper we adopt a rather more fundamental approach by developing a distribution for the faults over the software, this distribution then allows us to assess the effects of various testing strategies (and operational regimes) upon the failure-time statistics.

Besides determining a distributional family for the faults, one must also decide in what sense the faults are distributed over the software. In this paper it will be assumed that they are distributed over the logic paths through the software; a probability distribution for the faults over the paths can then be derived as follows.

Consider an item of software containing \( N \) faults randomly distributed over its \( M \) logic paths. Note that each fault will affect one or more paths; suppose that on the average, each fault affects \( c \) paths.

In the analysis contained in the following sections, we shall be concerned with the probability of locating an arbitrary fault in a randomly chosen path. We will work purely in terms of averages and assume that each fault actually does affect \( c \) paths. This is, of course, an approximation, but it avoids the difficulty of having to choose a probability distribution for the number of paths affected by an arbitrary fault and also it simplifies the analysis considerably. With this assumption, the distribution we seek can be obtained by assuming that the \( N \) faults are added to fault-free software in the following fashion.

Allocate the faults, one at a time, to \( c \) randomly chosen paths. Then, each time a fault is allocated, any given path receives a fault with probability \( c/M \). Thus, for any given path, the allocation of faults constitutes a series of \( N \) Bernoulli trials, with probability \( c/M \) of receiving a fault and probability \( (1-c/M) \) of not receiving a fault. Hence, after all \( N \) faults have been allocated, they are distributed binomially over the paths with

\[
\Pr(\text{An arbitrary path contains } \ell \text{ faults}) = \binom{N}{\ell} \frac{c^\ell}{M^\ell} \left(1 - \frac{c}{M}\right)^{N-\ell} \tag{2}
\]

and in particular

\[
\Pr(\text{An arbitrary path is fault-free}) = \left(1 - \frac{c}{M}\right)^N \tag{3}
\]

5. THE TEST PHASE

5.1 Distribution of Failure Times Under Random Testing

In this section, random testing is considered in the form of random selection and testing of paths.

If we consider the discovery of a fault in a particular path as an "event", the random testing of paths can be considered as a sequence of Bernoulli trials in which, prior to the removal of any faults, the probability of an event is \( 1 - \left(1 - \frac{c}{M}\right)^N \)

which follows from (3).

Now consider a unit time interval measured from the commencement of the random testing process. Let the probability of not locating a fault in this unit time interval be \( p \). Then, if testing is carried out at a rate of \( r \) faults per unit time, the probability of not locating a fault in a particular path is \( p^{1/r} \). Thus, from (3)

\[
p^{1/r} = \left(1 - \frac{c}{M}\right)^N \tag{5}
\]

Let the number of paths which have to be tested in order to locate a fault be \( B \). Then, since we are dealing with a sequence of Bernoulli trials, the expected value of \( B \) is

\[
E(B) = \frac{1}{1-p^{1/r}} \tag{6}
\]

and the expected time required to locate the first fault is

\[
\frac{1}{\lambda_1} = \frac{1}{r(1-p^{1/r})} \tag{7}
\]

Now, with random testing, the probability of locating a fault in any small time interval \( \delta t \) remains constant until the first fault is removed. This implies a constant failure rate between the fault-removal instants and hence an exponential distribution of failure times. The parameter of this distribution can be found by extending the above argument as follows.

Suppose that each path is split into \( k \) portions so that \( rk \) portions of paths are tested in unit time. Then, with \( p \) defined as before, the probability of not locating a fault in a given portion of a path is (prior to the removal of any faults)

\[
p^{1/rk} = \left(1 - \frac{c}{M}\right)^{N/k} \tag{8}
\]

The expected time to first failure may then be written

\[
\frac{1}{\lambda_1} = \frac{1}{rk(1-p^{1/rk})} \tag{9}
\]

and the parameter of the exponential distribution describing the time to first failure is given by

\[
\lambda = \lim_{k \to \infty} \frac{1}{rk(1-p^{1/rk})} \tag{10}
\]
where the subscript on $\lambda_0$ indicates that this is the failure rate when zero faults have been removed.

Equation (10) can be evaluated using L'Hôpital's rule to give

$$\lambda_0 = -N r \log \left(1 - \frac{c}{M}\right)$$

(11)

If the fault causing the first failure is located and removed, there will be $N-1$ faults remaining in the software. If each of the original $N$ faults were equally likely to be found, the remaining $N-1$ faults would obviously be binomially distributed over the paths. However, under our testing scheme, some faults are less likely to be detected than others. This is because we are selecting each path (for testing) with equal probability and some paths contain more than one fault. A path containing two faults is equally likely to be selected as one containing one fault, but under our scheme, only one fault will be removed after a failure. Thus, faults occurring in paths containing more than one fault are less likely to be eliminated (under our scheme) than those occurring in paths containing one fault only. Hence, after the first fault has been removed, the distribution of faults over paths is no longer binomial.

Fortunately, for all practical problems, the approximation involved in assuming that the distribution remains binomial as faults are removed is only very slight. In SRC software, the number of logic paths, $M$, is a very large number implying that the parameter $c/M$ of the binomial distribution in (2) is a very small number. This in turn implies that the number of paths containing more than one fault is a very small fraction of $M$. Consequently we can, to a very good approximation, assume that the distribution of faults over paths remains binomial as faults are removed from the software.

Thus, we will assume that removal of faults does not disturb the form of the distribution of faults over paths so that, after the first fault has been eliminated, the failure rate will be given by the right-hand side of (11) with $N$ replaced by $N-1$. In general, after $j$ faults have been removed, the failure rate will be

$$\lambda_j = -(N-j)r \log \left(1 - \frac{c}{M}\right)$$

(12)

This result indicates that under random testing with a binomial distribution of faults over paths, the failure rate is proportional to the number of faults in the software. This is in agreement with models of the JM-type and indicates that the proportionality factor $\phi$ can be written

$$\phi = -r \log \left(1 - \frac{c}{M}\right)$$

(13)

which, for practical software is a small positive quantity because, for such software, one would normally have $c < M$.

It is also worth noting that when $c < M$, (12) can be written

$$\lambda_j = (N-j)\frac{c}{M}$$

(14)

This equation is not really indicating that the failure rate is inversely proportional to the number of program paths, but rather it indicates that the failure rate is proportional to the number of faults per path.

As explained in [3] it is very easy to estimate the parameters $N$ and $\phi$ in equations (12) and (13) by the maximum likelihood method. These estimates allow a prediction of future mean times between failures (MTBFs) to be made; such predictions can be of value in deciding when the test phase should be terminated.

For the purposes of this paper, a simple graphical method of estimation, which has been found to give estimates very close to those obtained by maximum likelihood [12] will be explained.

From (12) and (13), we have

$$\lambda_j = (N-j)\phi$$

which implies that a plot of $\lambda_j$ against $j$ should give a series of points in a straight line with slope equal to $\phi$. This is indicated in Figure 1. Note that the line should cross the abscissa at $j=M$ so that an estimate of the initial number of faults in the software is easily obtained. Since an estimate of $\phi$ is given by the slope of the line, future behaviour is readily predicted.

5.2 An Improved Form of Path Testing

Throughout the random testing procedure of the previous section, each path was equally likely to be selected for testing regardless of whether or not it had already been tested. Thus, the random testing procedure can be considered as a case of random selection with replacement. Let us now briefly consider a slightly improved procedure corresponding to random selection without replacement. That is, we continue to select paths randomly, but we are clever enough to ensure that no path is selected twice.

From the commencement of this path testing strategy to the location of the first fault, the procedure can be represented by the semi-Markov model depicted in Figure 2. In this figure, state $M-1$ represents the state in which the $(1+1)$th path is being tested and state $F$ is the state of failure.

![Figure 1](image1.png)

![Figure 2](image2.png)

A good approximation to the behavior of this model can be obtained by assuming that the distribution of the transition time from state $M-1$ to state $F$ (given that such a transition takes place) is exponential with parameter denoted $\lambda_0(M-1)$ and given by

$$\lambda_0(M-1) = -N r \log \left(1 - \frac{c}{M-1}\right)$$

(16)

where the symbols on the right-hand side have the same meaning as in the previous section.

If a horizontal transition takes place (see Figure 2) it does so in deterministic time $1/r$.

The symbols attached to the arrows in Figure 2 indicate the transition probabilities of the underlying Markov chain and we have
An expression for the failure rate of this system can be obtained in a similar fashion to that employed in the previous section; i.e. split each path into \( k \) portions and then let \( k \to \infty \). The resulting expression for the failure rate function (prior to the removal of any faults) is

\[
\lambda_0(t) = -N_r \log \left(1 - \frac{c}{M - r t}\right)
\]

and, after \( j \) faults have been removed, we find

\[
\lambda_j(t) = -(N - j) r \log \left(1 - \frac{c}{M - r t}\right)
\]

which should be compared with equation (12).

Equation (19) represents a failure rate function which is monotonically increasing; i.e. this model leads to a failure rate which increases during the intervals between failures. Note, however, that large software systems usually have an astronomically large number of logic paths. Thus, at the end of the test phase, \( r t \) will be several (usually very many) orders of magnitude less than \( M \) and so the monotonic increase in the failure rate function (19) is quite negligible. In other words, this testing strategy would lead to times between failures whose distributions were indistinguishable from exponential distributions.

This particular form of testing strategy was discussed here partly to show that under a path testing strategy applied with reasonable uniformity over the software we can expect that software failure times will be exponentially distributed and partly to point out the main reason why, even after a rigorous testing and debugging phase, large software systems always contain a large number of faults. The problem is, quite simply, that large software systems are immensely complex (and this is why in (19) the number of paths tested, \( r t \), is always very much less than \( M \)). In the whole working life of such systems many faults will never be detected; others are detected so rarely and are so obscure that little or no attempt is ever made to rectify them.

Let us now turn to a method of testing which is distinctly more relevant to SPC systems.

### 5.3 Non-Uniform Testing

In the operational phase of SPC software, the code relating to call handling is much more heavily used than the maintenance software. In addition, large sections of the maintenance software are not critical to traffic handling. In recognition of these facts, it is usual for the code relating to call handling, and any other supporting software recognised \textit{a priori} as critical, to be tested much more thoroughly than the remainder of the code.

Testing of a large software system usually takes place in an incremental fashion; i.e. the system is not usually tested as a whole until after the various building blocks have each been subjected to a degree of testing. We will here confine our attention to the modelling of software failure behaviour following full system integration. The development in the previous two subsections could be applied equally well to a complete system or to a single module but we now wish to model behaviour under a non-uniform testing strategy.

As a first step toward a study of the effects of non-uniform testing, we can draw upon the techniques employed in the previous two subsections.

Assume that the software can be partitioned into two clearly-defined sections, one of which is heavily tested and the other one lightly tested. Refer to these sections as section 1 and section 2 respectively.

On any given test run, section 1 will be tested with probability \( p_1 \) (where \( p_1 > 0.5 \)) and section 2 will be tested with probability \( 1 - p_1 \). This is referred to as the testing profile.

Let \( N_1 \) be the number of faults originally in section 1 and let \( N_2 \) be the number of faults originally in section 2. Let \( M_1 \) be the number of logic paths through section 1 and \( M_2 \) the number of logic paths through section 2.

Assume that the faults have been distributed in the same fashion (but separately) over the two sections of software so that we will have, prior to any testing

\[
P(A \text{ path selected for testing is fault-free}) = \left\{ \begin{array}{cl} 1 - \frac{c}{M_1} & \text{if } i = 1, 2 \\ \frac{M_1}{M_1} & \text{if } i = 1, 2 \end{array} \right.
\]

so that, at the commencement of the (non-uniform) testing phase

\[
P(A \text{ path selected for testing is fault-free}) = p_1 \left(1 - \frac{c}{M_1}\right) + (1 - p_1) \left(1 - \frac{c}{M_2}\right)
\]

Following an argument identical to that used in section 5.1, we find that the failure rate prior to the removal of any faults is given by

\[
\lambda_0 = -r \log \left\{ p_1 \left(1 - \frac{c}{M_1}\right) + (1 - p_1) \left(1 - \frac{c}{M_2}\right) \right\}
\]

Prior to the removal of any faults, the probability that an arbitrary path through section 1 contains a fault is

\[
P_{1}(0) = 1 - \left(1 - \frac{c}{M_1}\right) \quad i = 1, 2
\]

where the argument in \( P_{1}(0) \) indicates that no faults have yet been removed.

Thus, under the assumed testing profile, when a fault is found, the probability that it lies in section 1 is given by

\[
P_{1}(0) = \frac{p_1 P_{1}(0)}{P_1 P_{1}(0) + (1 - p_1) P_2(0)}
\]

and the probability that the first fault is located in section 2 is given by

\[
P_{2}(0) = 1 - P_{1}(0)
\]

Thus, after removal of the first fault, the failure rate is given by

\[
\lambda_1 = -r \log \left\{ p_1 \left(1 - \frac{c}{M_1}\right) + (1 - p_1) \left(1 - \frac{c}{M_2}\right) \right\}
\]

\[
\lambda_1 = -r \log \left\{ p_1 \left(1 - \frac{c}{M_1}\right) + (1 - p_1) \left(1 - \frac{c}{M_2}\right) \right\}
\]
and, in general, after $J$ faults have been removed, an expression of the form below is obtained:

\[
\lambda_j = -r \log \left[ p_1 \left( 1 - \frac{c}{M_1} \right)^{N_1} - \sum_{k=0}^{J-1} \frac{c}{M_1} \left( 1 - p_1 \right) \left( 1 - \frac{c}{M_2} \right)^{N_2} - \sum_{k=0}^{J-1} \frac{c}{M_2} \right]
\]

Unfortunately, a simple analytic form for $\lambda_j$ does not appear to be available, but a recursive scheme for the computation of (28) is easily arranged.

It is worth noting that the argument of the logarithm in equation (28) is a sum of two terms. The first term relates to the heavily tested section and the second term to the lightly tested section. Recall that the heavily tested section represents that part of the software considered critical to traffic throughput. Thus, if the testing personnel accurately identify all areas of code which, in the presence of faults, could have a deleterious effect on traffic and test those areas thoroughly, only the first term in the argument of the logarithm in (28) has any significance as regards the effect of failures on traffic throughput. In such a case, the model effectively reduces (from the point of view of trafficability) to the one discussed in section 5.1. Unfortunately, because of the complexity of SPC software, it is not usually possible to identify all sections of code which could have a detrimental effect on traffic.

It is also worth noting that numerical computations have shown that equation (28) reflects a property commonly observed in the failure rate function of large software systems. Recall that models of the JM-type assume that the software failure rate is proportional to the number of faults in the software. This assumption implies that each time a fault is removed, the failure rate undergoes a step reduction with the size of the step being exactly the same for each fault removed. In his criticism of the JM-model, Littlewood [5] points out that faults located in sections of code which are heavily used should contribute much more to the software failure rate than those located in rarely executed sections of code. In addition, faults located in heavily used sections of code are much more likely to be located early. These two facts indicate that the step decreases in the failure rate function (which occur when a fault is removed) should be large in the initial stages of testing and gradually decrease as testing proceeds. This implies that if a plot is made of $\lambda_j$ against $J$ (as was done in Figure 1) a convex curve rather than a straight line should be obtained. Numerical computations based upon (28) have shown that a convex curve is indeed obtained.

6. TRANSFERENCE FROM THE TEST PHASE TO THE OPERATIONAL PHASE

The test phase is terminated when the testing personnel are satisfied that the software failure rate has reached a sufficiently low level that the software system can be released.

In section 5 the software testing problem was modelled in terms of the testing of a sequence of logic paths through the software and, at the end of section 5.2, attention was drawn to the fact that for large and complex software systems (such as SPC), the number of paths tested in a realistic testing period is bound to be extremely small in comparison to the total number of paths through the software. There are, of course, many ways of testing software besides path-testing (see, for instance, [13]), but it is a well-established fact that whatever testing scheme is used, time-limitations dictate that the aggregate of all input data sets employed during the testing period is bound to constitute an extremely small sample out of all possible input data sets. It is because of this that when SPC software is transferred from the test environment to an operational environment, this transfer is always accompanied by a (usually large) step increase in failure rate. Because the test phase can only consider a small sample out of all possible input data, it is impossible for the testing personnel to try out (or even think of) all the forms in which input data will be applied to the system in the various operational environments to which it will be subjected. Consequently, when an SPC exchange is exposed to real traffic for the first time, many new software faults are revealed and the failure rate increases dramatically above its level at the end of the testing phase.

This is a problem of considerable importance to the producers (and users) of SPC software. A major aspect of the problem stems from the fact that application of standard statistical techniques to failure data collected during the test phase is of no assistance whatever in attempting to estimate the step increase. Data collected during the test phase indicate a gradually decreasing failure rate and traditional statistical methods can do no more than predict a continuation of this trend.

An approach to the solution of this estimation problem can be developed from the technique of functional testing [13] which is widely applied to software systems in the later stages of the test phase. Large software systems are normally required to perform various functions, and it is quite common to test the ability of a software system to perform each of its required functions in turn. Each functional test will reveal a number of faults, and when the testing personnel are satisfied with the ability of the system to carry out a particular function, they commence a new functional test. The significant point to note is that implementation of a given function usually requires execution of a subset (often a small subset) of the software modules making up the overall system. In the testing of each such subset we can reasonably assume uniform execution of code so that a model of the JM-type ought to be appropriate. If we apply the JM model to the first subset of modules tested, the model will give us a prediction of the number of faults remaining in that subset when testing ceases. Noting that the various subsets tested as function testing proceeds will not be entirely disjoint, it should be clear that summing estimates of faults remaining in each subset should give an upper bound (estimate) on the total number of faults remaining in the overall software system at the termination of function testing. This is our approach to the problem of
estimating the step jump in failure rate which occurs when software is transferred from the test phase to the operational phase. We can illustrate it in terms of a set of software failure data which has appeared several times in the literature. The data are listed in Table 1.


**TABLE 1: SUCCESSIVE EXECUTION TIMES BETWEEN FAILURES IN SECONDS**

<table>
<thead>
<tr>
<th>Section</th>
<th>Data Points</th>
<th>1 to 29</th>
</tr>
</thead>
<tbody>
<tr>
<td>Section 2</td>
<td>Data Points</td>
<td>30 to 66</td>
</tr>
<tr>
<td>Section 3</td>
<td>Data Points</td>
<td>67 to 100</td>
</tr>
<tr>
<td>Section 4</td>
<td>Data Points</td>
<td>101 to 136</td>
</tr>
</tbody>
</table>

Recall the comments made above regarding function testing; it was pointed out that a model of the JM-type should be appropriate. Thus, a plot of failure rate against failure number should produce a straight line as in Figure 1. Note that Figure 3 commences with such a straight line and then undergoes an upward transition before following another straight line. There are two upward transitions and four straight line sections in all. This is precisely the kind of behaviour that one could expect if the test phase consisted of the testing of four separate functions in sequence and we will assume here that Figure 3 does, in fact, reflect such a testing scheme.

If we were directly involved with such a testing scheme, it would be a simple matter to record which data points correspond to which function. In our case we have to attempt to do this a posteriori and this we have done by a simple "eyeball" test on Figure 3 in conjunction with Table 1 (statistical tests were applied but did not provide results of sufficient significance to allow partitioning of the data). We partitioned the data into four sections as follows:

Let \( \lambda_i \) be the failure rate against failure number from the model of the JM-type and \( \phi_1 \) be the parameter of the model from which \( \lambda_i \) is estimated by a "rolling average" over 21 data points using the equation

\[
\lambda_j = \frac{21}{10} \sum_{r=10}^j d_{(r)} \]

where \( d_k \) represent the \( k \)th data point. The \( j \) values are spaced 5 apart in order to provide additional smoothing.

At first glance, it should be quite clear to the reader that application of any of the traditional statistical techniques to predict future behaviour (following that plotted in Figure 3) would predict that future failure rates for the system should be less than \( 10^{-4} \) and that there should be a steadily decreasing trend.

And yet, if the data plotted in Figure 3 represent failure data collected during the test phase of a software system (which, in fact, they do) then we know from experience that when this system is transferred to the operational phase, we can expect an increase in failure rate.

<table>
<thead>
<tr>
<th>JM parameters</th>
<th>No. Faults removed</th>
<th>Estimated No. Faults remaining</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \hat{N}_1 = 32 )</td>
<td>29</td>
<td>3</td>
</tr>
<tr>
<td>( \phi_1 = 4.20 \times 10^{-4} )</td>
<td>37</td>
<td>22</td>
</tr>
<tr>
<td>( \hat{N}_2 = 59 )</td>
<td>34</td>
<td>5</td>
</tr>
<tr>
<td>( \phi_2 = 5.50 \times 10^{-4} )</td>
<td>36</td>
<td>1</td>
</tr>
</tbody>
</table>

**TABLE 2**

If we assume that no software modules have been excluded from the test and note that some (if not all) modules would be required for more than one function, it follows that a summation of the entries in the right-hand column of Table 2 should give an approximate upper bound on the number of faults remaining in the software. Thus, an approximate
upper bound for the number of faults remaining in the software is $31$. This figure can now be used to determine an approximate upper bound on the step increase in failure rate which would be experienced in transferring this particular software system from the test environment to the operational environment. In order to do this, we need to estimate the proportionality factor $\phi$ for the operational phase.

It is noteworthy that the four groupings in Figure 4 show a monotonic decrease in $\phi$. Whether or not one could expect a further decrease in transference from the final stage of testing to the operational phase would depend very much on the manner in which the overall test had been carried out. Experience would no doubt prove useful in making a decision of this kind. Let us, for the sake of example, assume that it is deemed highly improbable that $\phi$ will be greater than $10^{-6}$ in the operating phase. Then, a reasonably safe upper bound on the failure rate at the commencement of the operational phase would be $31 \times 10^{-6}$.

Obviously there is room for a good deal of investigation into this important topic.

7. THE OPERATIONAL PHASE

In section 5.3 a model was developed which indicated that when software is nonuniformly tested, a plot of failure rate $\lambda_j$ against failure number $j$ describes a convex curve (under the assumption that each detected fault is immediately removed). After a (large) software system has been transferred from the test phase to the operational phase, a similar convex decline in failure rate is usually experienced. This fact can be attributed in part to nonuniform execution, but there are other factors at work also.

When a software system enters the operational phase, personnel are usually available who have considerable familiarity with the system but, because of the nature of the operational phase, it is unusual for faults to be rectified as soon as they are located. In addition, it is unlikely that the available personnel have the same familiarity with the software as those who were involved with the test phase. As time passes, there is an inevitable turnover in personnel leading to a decreasing familiarity with code and an increasing likelihood that attempts to eliminate faults will lead, instead, to the introduction of new faults.

This has certainly been the experience with the early SPC systems which were written in machine code. At the time that these systems were being developed, computer scientists were only just beginning to recognise the difficulties associated with complex software systems. As a consequence, these systems had, by today's standards, very little in the way of structured design and this, coupled with the fact that they were written in machine code, made the software extremely difficult to modify without the danger of introducing new faults.

The above facts ensured that the rate of decrease in failure rate, from the commencement of the operational phase, gradually reduced until the software exhibited a failure rate which was essentially constant. From this point on, the main problem for those charged with the responsibility of maintaining the software was to ensure that the failure rate did not begin to increase again in the face of the substantial changes which have to be made to the software every few weeks of operation (because of the introduction of new facilities). By and large, they appear to have been successful and failure rates have been held at an acceptably low, though essentially constant, level. From an analytical point of view, this is most encouraging because a constant failure rate implies that times between failures are exponentially distributed.

Less is known about the reliability performance of the more recently developed SPC software systems. These systems have mostly been produced using the ideas of structured design [11] and as such, should not only contain considerably less faults than the early systems but should also be far easier to maintain (note that in relation to software the term "maintenance" relates to any modifications made after commencement of the operational phase). Consequently, it is quite possible that the failure rate function will show a monotonic decrease throughout the operational phase, although it seems more likely that the forces which worked against the earlier systems will eventually work against the newer systems. However, if the failure rate of the newer systems does eventually level off, it will surely do so much later in the operational phase and at a much lower level.

Although these facts indicate that SPC systems are likely to exhibit increased reliability in the future, they also indicate, when coupled with the effects of nonuniform execution, that an exponential distribution may not provide a very good fit for software failure time intervals during the operational phase. As a consequence, considering that very little data are available on these systems up to now, it would seem to be a useful exercise to try and extend the models developed in section 5 to represent behaviour in the operational phase. We have made some tentative steps in this direction; three of these are presented below.

7.1 Modelling Delayed Fault Removal

In the models developed in section 5, we assumed that each fault detected was removed before the software was executed further. Since we are measuring time in terms of computer run-time, this assumption implies that a fault which is detected is instantaneously removed. We pointed out above that in the operational phase it is not usually possible to rectify faults in this way, and a simple model of "delayed" fault removal can be developed as follows.

Assume that when the $i$th fault is detected, there is a probability $p$ that it will be removed instantaneously. Assume further that if it is not removed instantaneously, there is a probability $p$ that it will be removed at the instant of the $(i+1)$th failure; if it is not removed at the $(i+1)$th failure instant, it is removed with probability $p$ at the instant of failure number $i+2$; and so on. Let these assumptions be made for each fault.

Denote the expected number of faults removed at the $i$th failure instant by $R_i$ and let $q = 1 - p$. Then we have

$$R_i = p \sum_{k=0}^{i-1} q^k = 1 - q^i$$
This result indicates that delayed fault removal has only a slight, and rapidly diminishing, effect on the failure process. As \( i \) increases, the expected number of faults removed at the \( i \)th failure instant rapidly approaches unity.

7.2 Imperfect Debugging

In order to develop a simple model for imperfect debugging, let us assume that no more than one fault is ever rectified at any one failure instant. Let us further assume that when fault-removal is attempted, there is a probability \( q^2 \) that \( r \) faults will be inserted instead of a fault being removed. If we let \( r \) range over the positive integers, this implies a probability \( 1-q/p \) that the fault is removed (where \( p = 1-q \)).

Denote by \( R \) the expected number of faults removed when such removal is attempted. Then we have

\[
R = 1 - \frac{q}{p} - (q + 2q^2 + 3q^3 + \ldots) \quad (31)
\]

Thus, the failure rate function will be non-increasing (i.e. the "system" will be stable) so long as \( R3q \), i.e. so long as

\[
\frac{a}{p} (1 + \frac{1}{p}) < 1 \quad (33)
\]

This condition implies that the system will be stable so long as the probability of correctly removing a fault is, at each attempt, greater than (or equal to) \( 2 - \frac{1}{\sqrt{2}} = 0.586 \).

This model can be fairly readily extended to take account of the fact that in practice, the probability of incorrect fault removal (i.e. insertion of extra faults) tends to increase with time. Such extension seems to indicate that the condition for stability (i.e. \( \Pr \) fault correctly removed) \( \geq 0.586 \) remains the same.

It is worth noting that there have been cases where the stability condition has not been met. For obvious reasons, these cases have not been widely publicised but some discussion of this fact appears in Lehman [14] who terms the phenomenon "software fission".

7.3 The Operational Profile

In section 5.3 a recursive relationship was derived for the failure rate of a software system in which part of the system was tested more heavily than the remainder. The "testing profile" was defined by a probability \( p_1 \); which effectively represented the proportion of total testing time which was spent on the heavily tested portion. In a similar fashion, an operational profile can be defined; in the simplest case, we can again split the software into two sections, one of which is heavily used, the other less so. A simple profile of this kind should reasonably reflect the usage of SPC software in the operational phase: those parts of the software associated with call handling are heavily used and the remainder is less heavily used.

If we ignore for the moment the existence of the step increase in failure rate which occurs on transferance of software from the test phase to the operational phase, an expression similar to (28) can be written down for the failure rate to be expected in the operational phase.

Suppose that in the operational phase the heavily-used parts of the software are executed a proportion \( p_2 \) of the time. Thus, \( p_2 \) defines the operational profile which need not be the same as the testing profile defined by \( p_1 \). (Indeed, an interesting question, which will be briefly discussed below, concerns the optimum relationship between \( p_1 \) and \( p_2 \).)

Consider a software system whose parameters are identical to those of the system discussed in section 5.3. Suppose that this system commences its operational phase after \( j \) faults have been removed under testing profile \( p_1 \). Then, the failure rate under operational profile \( p_2 \) is given by

\[
\lambda_{ops} = -r \log p_1 (1 - \frac{q}{M_1} - \frac{q}{M_2} - \ldots) \quad (34)
\]

Adjustments to this formula can be fairly easily made to take account of (i) the fact that the software which is heavily used in the operational phase may not be identical to that which is heavily tested (e.g. some maintenance software may be heavily tested because it has the potential to disrupt call-handling) and (ii) the fact that some of the lightly-tested software may have negligible effect on call-handling.

Let us turn briefly now to the question of the optimum relationship between \( p_1 \) and \( p_2 \). Obvious questions which spring to mind are (i) should one devote all one's testing effort to those sections of the software which will be heavily used and simply fix faults which are detected in the other sections during the operational phase? (ii) should we instead have \( p_1 = p_2 \)? Numerical experiments with (34) indicate that neither of these proposals constitutes the optimum solution in general, although (i) appears to be optimum if a relatively high failure rate can be tolerated. Further investigations are required here; any results obtained in this direction should be at least approximately valid even though we have ignored the "step increase" phenomenon.

REFERENCES


Session 2.3

ITC-10

paper #2
Q.1 (Lars Peiram)

To my own opinion the adoption of probalistic and statistical methods to the analysis and prediction of Software Reliability goes on very slowly. Perhaps we should ask ourselves if we are approaching the matter along appropriate times. Your paper, being very interesting, raises two questions in this spirit:

1. Is there not a fundamental ambiguity in the definition of a "software fault"? To my experience the underlying causes for a "software failure" are open complex and not always fully understood. With this in mind, is then "the number of faults" a sound model parameter?

2. Considering the relatively slow-rate with which software failures occur, a) compared with the rate of software changes and other changes in the environmental, is exponentiality of the inter-failure distribution of substantial importance?

A.1 (T. Downs)

1. In answer to your first question, I think you will concede that software reliability must depend in some way upon the number of faults in the software and so this number (or some function of it) must be accounted for in the development of a realistic model. I certainly agree that the actual number of faults in the software does not necessarily give a good guide to reliability performance and this has been discussed by Littlewood (in my ref. (5) and in some of his other publications). In my modelling I have used the failure rate function as a measure of software reliability and, of course, because of the simple assumptions I have made, the number of faults appears as a parameter in each expression. With experience, correction factors could, if necessary, be introduced to account for the more obscure types of failures.
A.1 (Cont'd.)

2. This is a good point, but it does not invalidate my conclusion (not stated in the paper, but stated at the conference) that because some sections of SPC software are more heavily tested than others, the failure rate fluctuates randomly between various levels. This implies that the times between failures are not exponentially distributed (see D.P. Gaver, Technometrics, 1963). Regardless of the frequency of changes made to the software, a fluctuating failure rate is bound to remain. If the inter-failure distribution were exponential it might be possible, on the basis of experience, to predict the effects of changes by simple Markov analysis. Numerical experiments are required in order to determine how far from exponential the distribution may be.

Q.2 (D. Manfield)

In your models you calculate failure rates based on the number of logical paths in the software. In complex modular software the code may be correct in each module, but many failures may occur because of incorrect interaction of different modules. Can your approach be used to consider the logical relationship between modules?

A.2 (T. Downs)

My modelling approach is essentially "macro" modelling and aimed at examination of the performance of a fully-integrated system in the form of a black box. Because of this the model does not distinguish between different types of failure.

The problem of modelling failures due to interactions between different modules has been investigated in at least two papers. The major difficulty in applying such models in the enormous amount of data which is required. Most of this data is required to estimate transition rates between modules.

Q.3 (J.R. De Los Mozos)

In the article you distinguish between sections of the system SW which may have different importance from the testing viewpoint, such as call processing and maintenance SW. From the traffic viewpoint I realize other possible approaches. For example I think that depending on the number of calls affected by the fault may influence the fault removal process. If that were the case we could assume that, after sometime in the operational phase, the number faults remaining in the SW will very much depend on the effect that each type of fault has on call handling. Are you considering this aspect in your future research?
A.3 (Cont'd.)

I have presented a very basic approach to the modelling problem in which I have attempted to avoid any significant mathematical difficulties. In other words all the models described are quite simple in concept.

There is no doubt that these models could be made more realistic in many ways. Your suggestion is a welcome one and I will probably look into it.