Software Testing and Reliability for Telecommunication Systems

Wohlin, C.

**In Software Engineering ´86, pp. 27-42,
edited by D. Barnes and P. Brown,
Peter Peregrinus Ltd, United Kingdom, 1986.**

Software Testing and Reliability for Telecommunication Systems

by

Claes Wohlin, Lund Institute of Technology, Sweden

SUMMARY

This paper deals with software testing, and how to estimate reliability through models during testing. An existing classification of software reliability models are described. The existing models do not work during function testing of telecommunication systems and due to this an investigation of a couple of projects was made. A Markovian model for prediction of software error occurrences is developed and the use of the model is described in three parts; early use, main use and further development. Results obtained from the model are compared with software error data from another telecommunication project, than those investigated. The necessity of a mathematical and scientific foundation of Software Engineering is stressed.

SOFTWARE TESTING AND RELIABILITY FOR TELECOMMUNICATION SYSTEMS

Claes Wohlin
Lund Institute of Technology, Lund, Sweden

1. INTRODUCTION

One aim in Software Engineering must be to make software testing unnecessary, i.e. we have to develop error free software. This calls for better methods and tools during specification, design, coding and maintenance. However, today we are far from this ideal situation. Instead 30-50% of the total development effort is spent on software testing (verification and validation), see Boehm (1).

Without exaggeration we can claim that:

Software testing is today our best weapon against software errors, which makes testing an important (and necessary) part in both developing and maintaining reliable software.

Before we discuss software testing any further, we must be aware that the goal of software testing is to find errors and not to prove the absence of them.

We will here consider telecommunication systems, but the ideas are applicable to all other types of systems and the specific results are applicable for systems with a similar structure to those considered. The main point of the paper is not the model itself, but the way it is possible to investigate an environment and develop a model suitable for that specific environment.

Telecommunication systems are often built in a hierarchical structure and as the project progresses we apply different test techniques, e.g. unit testing, function testing, system testing and field testing. These are applied during different phases in the development process. The test techniques are, of course, dependent on the system structure, but if we consider a system with hierarchical structure, then these techniques are applicable. It is important to understand the different test techniques and to know what to expect from them and to be aware of the relations between them. We will only be concerned here with function testing. This form of testing can be described as follows;

Function testing

This is the verification of a specific function performed by the system. A function often involves more than one unit, see Fig.1.1. A unit is often a program module. We

have in this case six units and we consider three different functions. This form of testing includes what we call integration testing, e.g. the verification of interfaces between the system parts. During function testing the system grows unit by unit until we finally obtain the complete system.

Expectations: To find errors caused by problems arising when different units communicate and influence each other, and errors introduced during the design phase.

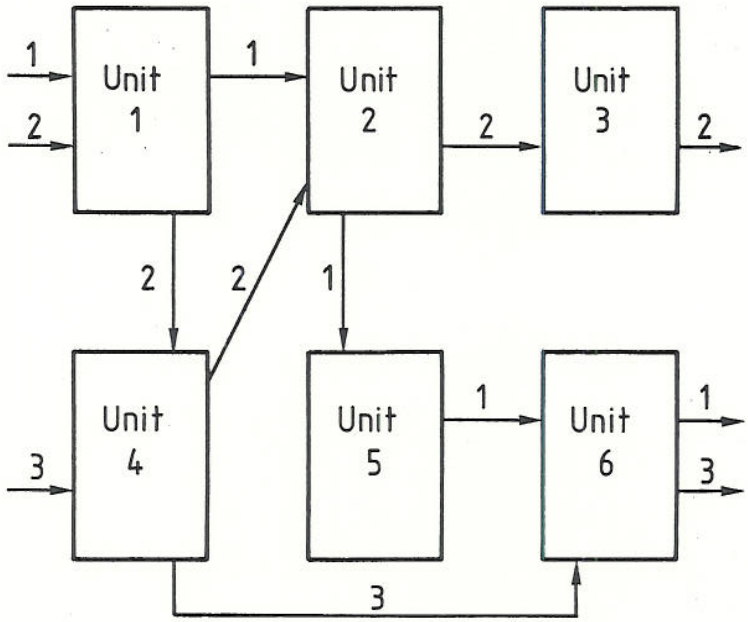

Fig.1.1 An example of how the units can communicate, when a number of functions are performed

The work in this paper emphasizes the development of a software reliability model*, which is used during function testing or similar test methods, and to show how it can be used to control the development of our software product. The existing models are not applicable during this early phase of testing, due to the assumptions made when developing them, (see below). The need of a model during function testing led to an investigation, which will be described shortly in this paper. For more information on the investigation see Wohlin (2). The investigation leads to a software reliability model for function testing. The model will take the stochastic behaviour of software error occurrences into account.

*When we use the word model, we mean a software reliability model in general. If something else is meant, it will be indicated.

2. SOFTWARE RELIABILITY MODELS

The reliability of a software product is one of the most essential measures of its quality. The traditional definition of reliability is applicable to software too.

> Software reliability - the probability that software will not cause the failure of a system for a specified time under specified conditions, see Goel (3).

A first prediction of the product's reliability could be obtained early in the project by estimating the correctness, through complexity measures. For fuller information on this see Lennselius (4). We are, however, concerned here with estimating the reliability while testing the product. We can obtain an estimation of the reliability through parameters as mean time to failure (MTTF) or number of remaining errors.

Independently of how we define a software error, applying more testing should reduce the number of remaining errors and thereby make the software more reliable. We would now like to know how reliable our software product is, and to obtain a measure of this, we have to have a software reliability model. There are a number of models presented in the literature and according to reference (3) the software reliability models can be placed in four classes. This classification is made on the basis of the failure process studied. Goel defines the following classes:

1. "Times between failures" models
   Explanation: The general approach for this class is to assume that the time between failure number (i-1) and failure number (i) follows a distribution, whose parameters depend on the number of errors remaining. One of the first and most commonly used models is the Jelinski-Moranda De-Eutrophication Model (5).
2. "Failure count" models
   Explanation: This class of models assume that the number of detected failures in an interval follows a stochastic process with a time dependent discrete or continous failure rate. A well-known model from this class is the Goel-Okumoto Non-Homogenous Poisson Process Model (6).
3. "Fault seeding" models
   Explanation: In this class we "seed" a known number of errors in to our product and when testing the product we find both seeded and unseeded errors. If we look at the proportion of seeded errors found compared to the number of unseeded errors found, we can estimate the total number of errors in the software product. The most widely spread model of this class is probably Mills' Seeding Model (7).
4. "Input domain based" models
   Explanation: The basic approach in this class is to generate a set of test cases from a distribution. This should be chosen so that it is representative of the operation of the software product.

Most developed models fall into classes 1 and 2, and the author feels that this is no coincidence. These two classes are time dependent, which gives us an opportunity to estimate the forthcoming software error occurrences. Classes 3 and 4 only give us a stationary value, e.g. the initial number of errors in the software product, but we do not get any information on how they will (hopefully) decrease. The different classes of models will be suitable during different times in the project and applications. We will, however, not consider this problem here.

The classification does not say that there is no connection between the classes. It is well-known from probability theory and queueing theory, (Kleinrock (8)), that there is a relationship between the distribution for the time between two consecutive events and the distribution of the number of events in an interval. This fact is further discussed in Wohlin (9), where this relationship is used to compare the models developed in references (5) and (6).

Two software reliability models, see references (5) and (6), have successfully been used in earlier telecommunication projects and the results were presented by Vrana and Wallander (10). They adopted, adjusted and applied the models to several projects, during the operation phase of the system. Encouraged by their success these models were applied to new projects. Unfortunately they did not work.

The difference, between the new applications and the ones made earlier, was that in the new projects the models were applied early, during function testing. The problem during this phase is that some of the assumptions made for the models are far from fulfilled. There are, however, some assumptions we accept and some we do not accept. One assumption we probably have to accept is an assumption made by almost all models. They assume either that the times between failures are independent or that the number of errors detected during non-overlapping intervals are independent. This independence is mostly not true, but experience shows that this violation does not in any significant way affect the results. Without assuming independence in probability theory we soon end up with very complex problems that are difficult, if not impossible, to solve.

We wanted a software reliability model for testing done in parallel, i.e. function testing, on a system with hierarchical structure. That is, we wanted a model for a given system structure and test environment. We could not, however, find any suitable software reliability model that fulfilled our requirements and therefore we had to develop a model of our own. But before we could do this we had to examine how the function testing is done in more detail. While examining this we will be able to identify the assumptions violated. And when we have identified the violations, we have the possibility of developing a more suitable model for function testing than the existing models.

3. THE INVESTIGATION

The problem outlined above led to an investigation. A couple of different telecommunication projects were investigated: we talked to managers, programmers, test groups etc. and tried to get a picture of what happens during the different test stages, especially during function testing.

We will only present here those parts of the investigation that are of interest in developing the software reliability model for function testing. For fuller information about the investigation and the results of it see reference (2).

In Fig.3.1 a simplified picture of the function testing environment is shown. This figure shows the main parts involved and how they communicate. The numbers and arrows in the figure call for some explanations.

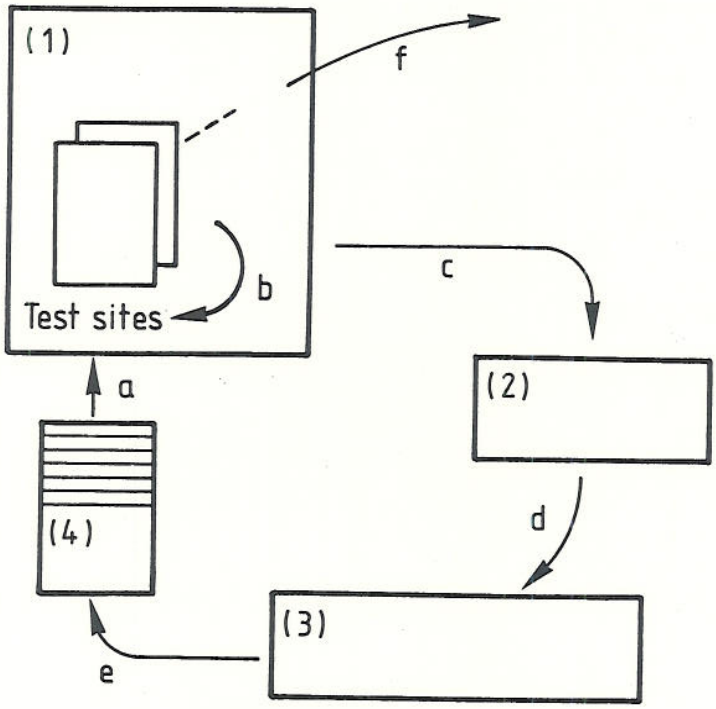

Fig.3.1 A simplified picture of the function testing environment

Parts

(1) We assume a special testing department, with one or many test sites depending on the size and cost of the system or project.

(2) An error report is written for every error that is found. The report is filed at a department for error collection.

(3) One department is occupied with construction and developing, i.e. programming.

(4) The fourth part is a model: We assume that when a unit

is ready it is put aside. And when all (see Fig.1.1) units for a function test are ready, it is put into a queue. The test remains in the queue until a test site is ready to accept it for testing.

Information flow

(a) When a test site is empty and the queue is not, test preparations are made and the test started.
(b) If an "easy" error is found the tester makes a correction, writes an error report and continues the test.
(c) The error report is sent to a special department for filing.
(d) If it is a "difficult" error, the tester stops the test, writes an error report and starts a new test if there are any tests waiting. If the queue is empty, wait until it is not. All error reports always go back to the programmer responsible. If it is an "easy" error the programmer should either approve of the correction or make a new one. And if it is a "difficult" error the programmer is responsible for correcting the error.
(e) A new or corrected unit is put aside and, when suitable, put into the queue, see point (4) above.
(f) The function test is done according to the test specification and the tester's ingenuity. When all (found) errors are corrected, it is considered completed.

We are now able to pin-point the assumptions which make the existing models fail, and they are:

1. Testing is representative of the operational usage, (or the tests are at least made on one test site, one after the other).
2. A detected fault is immediately corrected.
3. The failure rate is proportional to the number of remaining errors or decreases with test time.

For fuller information on these and other assumptions see reference (3).

The assumptions mentioned above are not valid, because:

1. The function testing is usually done at more than one test site at a time, which means that we have different tests going on in parallel.
2. Depending on the severity of the error, it is either corrected by the tester or, after a while, by the programmer.
3. Since we are testing different functions in different tests, the failure rate will not decrease in all tests just because we find an error in one function.

The third point is very important, because the failure time distribution is an important parameter when predicting the number of remaining errors etc. This is shown in Wohlin and Vrana (11).

From the information obtained here, it is possible to develop a software reliability model that is more suitable for function testing than any other existing model known by the author.

4. MODEL DEVELOPMENT

Now we have a background and therefore it is possible to develop a software reliability model. This model will be based on mathematics, probability theory and transform theory. For those who need an introduction or additional information on these subjects we recommend Feller (12) and reference (8).

Before we can do any mathematical derivations, we have to make some assumptions.

Assumptions for the model

1. Independent times between failures.
2. The rate of finding an error is constant and equal for all test sites at all times.
3. The number of test sites is constant and each test site can be in four different states:
   a) Searching for errors.
   b) Correcting an "easy" error and writing an error report on it.
   c) Writing an error report on a "difficult" error or finishing a test and then starting a new one.
   d) The same as c), but waiting for a function test to be ready.

   The times spent in state b)-d) (t_1, t_2 and t_3) are the mean values of the times in the different states.
4. We assume that 3. b)-d) can be weighed together to a mean value and that this value gives us a rate for correction of errors, which is constant and equal for all test sites at all times.

With these assumptions it is clear that a test site can be in one of two states; that is, searching or correcting errors. Both these states are left at a constant rate, not necessarily the same rate. This is equivalent to the fact that the times in each state are exponentially distributed with mean value 1/(the rate to leave the state). That is, each test site works as a two-state machine. The stochastic process involved when the times in a state are exponentially distributed are often referred to as a Markov process. The problems arising for this type of process can be solved with the methods developed for Markov chains. The Markov process is central both in queueing and reliability theory. The process is characterized by its "memoryless" properties.

The following notation will be used:

$P(\ )$	- the probability that ( ) is true
$F(t)=P(\tilde{t}\leq t)$	- the probability distribution function (PDF)
$f(t)=dF(t)/dt$	- the probability density function (pdf)
$F^*(s)$	- the Laplace transform of f(t)
$E(X)=\bar{x}$	- the mean value of X
$E(X^k)=\bar{x^k}$	- the k:th moment of X
$V(X)$	- the variance of X
M	- the number of test sites

state k - k test sites in correction state and M-k in searching state

X_k - the time in state k, (pdf f(t,k))

Y_k - the time until an error is found, when we are in state k, (pdf g(t,k))

Y - the time until an error is found, from an arbitrary time, (pdf k(t))

λ - the rate of finding an error for a test site

μ - the rate of correcting an error for a test site

Using assumption (4) from above, we obtain

$$1/\mu = P(\text{"easy" error})*t_1 + P(\text{"difficult" error or finished and new test available})*t_2 + P(\text{"difficult" error or finished and wait})*t_3 \quad (4.1)$$

As we said before, the time each test site spends in its two states is exponentially distributed, that is if we let u(t) be the pdf for the time in searching state and v(t) be the pdf for the time in correction state, we obtain

$$u(t) = \lambda \exp(-\lambda t) \text{ and } v(t) = \mu \exp(-\mu t) \quad (4.2)$$

We can now draw a Markov chain (see Fig.4.1), where state k is defined above. And knowing from the mathematics, that the merging of many exponential distributions still gives us an exponential distribution, where the rate is equal to the sum of the rates from the merging distributions, we obtain

$$f(t,k) = (k\mu + (M-k)\lambda) \exp((-k\mu + (M-k)\lambda)t) \quad (4.3)$$

that is the pdf for the stochastic variable X_k.

To be able to continue our derivations, we need to know the probabilities of being in state k and go to either state (k-1) or state (k+1) and we also need the stationary probabilities of being in state k. Let q(k,k-1) be the probability to go to state (k-1) if we are in state k, and p(k) be the stationary probability of being in state k.

From Figure 4.1 we immediately obtain

$$q(k,k-1) = \frac{k\mu}{k\mu + (M-k)\lambda} \text{ and } q(k,k+1) = \frac{(M-k)\lambda}{k\mu + (M-k)\lambda} \quad (4.4)$$

It is a little more work to obtain p(k), but there are methods for this presented in the literature, reference (8) and (12). One method is to make a cut through the Markov chain and study the flow across the cut, doing this we obtain

$$k\mu p(k) = (M-(k-1))\lambda p(k-1) \quad (4.5)$$

This can be solved recursively in k, and we find that

$$p(k) = \binom{M}{k} \left(\frac{\lambda}{\mu}\right)^k p(0) \quad (4.6)$$

p(0) is obtainable, because we know that

$$\sum_{k=0}^{M} p(k)=1 \tag{4.7}$$

That is

$$p(0)=1/\left(\sum_{k=0}^{M}\binom{M}{k}\left(\frac{\lambda}{\mu}\right)^{k}\right)=1/\left(1+\frac{\lambda}{\mu}\right)^{M} \tag{4.8}$$

Finally we obtain

$$p(k)=\binom{M}{k}\left(\frac{\lambda}{\mu}\right)^{k}/\left(1+\frac{\lambda}{\mu}\right)^{M} \tag{4.9}$$

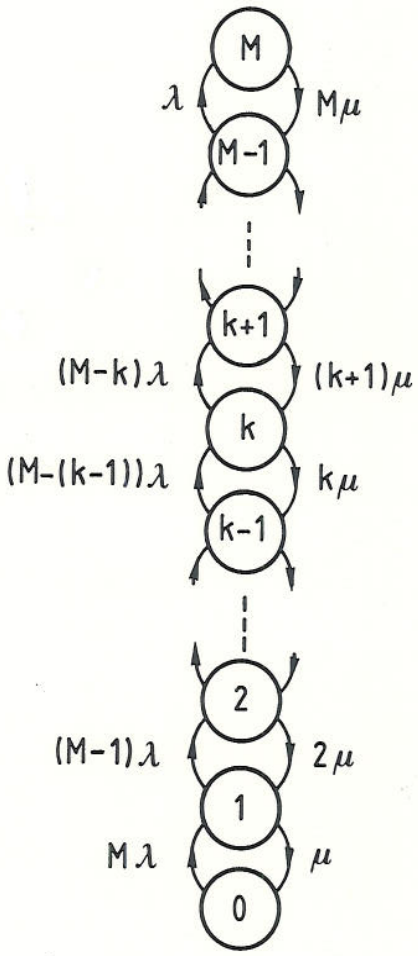

Fig.4.1 A Markovian chain for the test sites

We can find a relationship directly from the notation and by the use of the transition probability q(k,k-1), between some of the times considered, that is

$$Y_k = X_k + q(k,k-1)Y_{k-1} \tag{4.10}$$

A valuable property of the Laplace transform is that, when we have a sum of stochastic variables we obtain a product of transforms. If we use this property and knowing the pdf (f(t,k)) of X_k, with Laplace transform F*(s,k), we obtain a recursive formula in G*, where G*(s,k) is the Laplace transform of the pdf (g(t,k)) of Y_k

$$G^*(s,k) = F^*(s,k) \cdot G^*(q(k,k-1)s,k-1) \tag{4.11}$$

where

$$G^*(q(0,-1)s,-1) = 1 \tag{4.12}$$

Solving this recursively, we obtain

$$G^*(s,k) = \prod_{j=0}^{k} F^*\left(s \cdot \prod_{i=j+1}^{k} q(i,i-1), j\right) \tag{4.13}$$

Let

$$a(j,k) = \prod_{i=j+1}^{k} q(i,i-1) \tag{4.14}$$

We define Y as

$$Y = \sum_{k=0}^{M} p(k)Y_k \tag{4.15}$$

Applying the Laplace transform on this

$$K^*(s) = \prod_{k=0}^{M} G^*(p(k)s,k) \tag{4.16}$$

And finally we arrive at

$$K^*(s) = \prod_{k=0}^{M} \prod_{j=0}^{k} F^*(p(k)a(j,k)s,j) \tag{4.17}$$

where p(k), a(j,k) and F*(s) are easily found.

5. THE MODEL AND ITS USE

Let us stop here and consider what we have obtained through the derivation. The result from the derivation was a Laplace transform of the stochastic variable for the time it

takes to find an error from an arbitrary time. The transform can be inverted numerically, by methods found in, for example, Karlsson and Stavenow (13). It is, however, probably more interesting to find the mean and variance of the time until next failure is found. This can be done by using the moment generating properties of the Laplace transform, see reference (8).

That is, we have derived a formula for calculating some interesting parameters during function testing, or similar tests, where we have many tests going on in parallel.

If we now turn to the use of the model, we can divide it into three parts: early use, main use and further development. Let us consider them one by one:

Early use

1. λ and μ are decided through experience and knowledge of the methods to be used in the forthcoming project.
2. We can at this early stage, even before the project is started, test different values on M and see what happens. Supposing that the initial number of errors (N_0), when the function test starts, is known through experience, we can find the mean and variance of the time to remove a certain number of them. This will give us an opportunity to see if it is at all possible to meet the expected target date or if we have to invest in more test sites.

Main use

1. The project, on which the model is to be applied, has now started and the methods for correcting errors are well-known. This means that we can estimate t_1, t_2, t_3 and the probabilities concerned to calculate μ. We can also obtain an estimate of the initial number of errors through estimations from complexity measures, see reference (4) and (10).
2. When the function test is started, we record the times between failures. These times are compared with the calculated mean value from the model, and by the use of the principle of least squares, we can get the best possible estimate of λ. The estimation gives us the possibility of calculating some other interesting parameters, (see below). λ can be reestimated as the function test continues and we obtain more data from the test.
3. We can, for example, calculate a new estimate of the initial number of errors (N_0). This estimation can be compared with the one obtained from the complexity measures. We would like to suggest an easy formula for estimation of N_0, that is

$$N_0 = C \cdot \lambda \qquad (5.1)$$

where C is a constant, known from earlier projects.
4. The forthcoming behaviour of the stochastic process that rules the error occurrences can be estimated, e.g. the mean value and variance for the time until the next failure occurs and thereby the same for the time until a

certain number of failures have occurred. This problem is further discussed in (11).

5. The estimation of λ can also be used in simulations of the stochastic process mentioned above.
6. We have now (hopefully) obtained an improvement of the estimation of N_0 and, of course, we know how many errors we have removed, which means we also know the number of errors remaining when the system test starts. There is now an input value to another software reliability model, which is more suitable for system testing and operation, see for instance (5), (6) and (10).

Further development

1. We can introduce correction factors: factors that reflect feelings rather than facts. The project manager should try to compare a number of factors with what we can call a "normal" project.
   Some example of factors:
   1) How well is the task of writing error reports carried out?
   2) How experienced are the people involved?
   3) What is the rate of rotation?
   4) How is the project influenced by other projects?
   5) How is the project influenced by holidays, courses, conferences etc.?

   The list of factors can be made much longer. We would like to measure these factors in some way, for instance in percentage. Suppose that 50% is normal and over 50% is better than normal and consequently less than 50% is worse than normal.
   Let us call the measure of factor (i) for $\alpha(i)$ and we can also introduce weighing factors c(i), that is if we feel that one factor is more important than another we can give this c(i) a higher value.
   We would like to suggest that we can estimate the mean time to failure (MTTF) as

$$MTTF(corr) = MTTF \cdot \sum_i c(i) / (2 \sum_i c(i)\alpha(i)) \qquad (5.2)$$

   In a "normal" project MTTF(corr) is equal to MTTF.
   Correction factors call for a very fair judgement by the project manager.
2. The experiences, from this Markovian model for function testing, can be used, when developing a more detailed model. It will probably be possible to treat a company as a queueing network, similar to a computer network with nodes and packets. This is if we let, as an example, the departments be the nodes and the documents the packets. We can in this way obtain a very detailed picture of what happens in the company. This also gives us an opportunity to identify bottlenecks and test different changes in the organization in theory, before we implement it into our environment. The possibilities of an analytical solution to a model of this type are small, but we can simulate the model in most cases.

We have now discussed the use of the developed model, but there are other factors to be considered before we start using a software reliability model at all. Some of the most important ones are:

1. It is essential that we do not take models developed for other applications and try to squeeze them into our environment and needs. We have to investigate the assumptions made for the models and compare them with our environment before we decide if they should be used.
2. All information about our projects must be thoroughly and objectively recorded. Perhaps we even have to have a seperate metric group as proposed by DeMarco (14).
3. We believe that one model is probably not enough, what we need is different models during different phases of the software life cycle. As pointed out above and shown in (10), existing models may well apply if we look at the whole life cycle or just the operation phase.

6. RESULTS

This is an example of the use of the developed model:

To be able to use it we have to derive formulas for the mean and variance of the time until the next failure is detected. This is done from formula (4.17) and by using the moment generating properties of the Laplace transform, see reference (8) and (12).

By taking the two first derivatives on K*(s) and then letting $s \to 0$, we obtain

$$E(Y) = \sum_{k=0}^{M} p(k) \sum_{j=0}^{k} a(j,k)\bar{x}_j \tag{5.3}$$

and

$$E(Y^2) = \sum_{k=0}^{M} (p(k))^2 \cdot \sum_{j=0}^{k} (a(j,k))^2 \overline{x_j^2} + \sum_{k=0}^{M} p(k) \sum_{j=0}^{k} a(j,k)\bar{x}_j \cdot \sum_{r=0}^{M} p(r) \cdot \sum_{i=0}^{r} \begin{cases} 0 & \text{if } (k=r) \text{ and } (j=i) \\ a(i,r)\bar{x}_i & \text{else} \end{cases} \tag{5.4}$$

From these two we are able to calculate the variance, since

$$V(Y) = E(Y^2) - E(Y)^2 \tag{5.5}$$

The mean value can be found quite easily directly from Fig.4.1. A closer look at the mean value, shows that it satisfies our intuitive interpretation of what it ought to be, if we take all possibilities into consideration.

Here we will only consider the main use of the model, (see above). The results in Table 6.1 have been found by:

1. Estimation of the mean number of test sites in use, M, and of the correction rate, μ.

2. Collection of failure time data.
3. We use the collected data to find the best possible estimate of the failure rate, λ .
4. The value of λ is now used in formulas (5.3) and (5.4) to obtain the mean and variance of the time until the next failure is found.
5. We can also calculate the mean and variance until a certain number of errors has been detected, since
   the mean time to N errors = $N \cdot E(Y)$
   the variance in the time to N errors = $N \cdot V(Y)$
6. This gives us an opportunity to calculate a confidence interval for the time until a certain number of errors has been found. A 95% confidence interval is shown in Table 6.1. The number of errors used for prediction is shown in the left column.
7. As the function test proceeds we collect more and more data, i.e. goto 2.

TABLE 6.1 - 95 % confidence interval for prediction of the time until error number (i) has been found.

The time until error number (i) was found	79	119	149	195
Error number (i)	50	70	90	110
Number of errors used for prediction	95 % confidence intervals			
10	60.1-89.9	86.7-123.3	113.9-156.1	141.4-188.6
50	-	99.4-121.8	126.3-158.1	154.3-193.3
70	-	-	140.1-164.7	168.5-203.1
100	-	-	-	190.8-209.6

The results show that we are able to get a good picture of how errors are detected during function testing by use of the developed model. The predictions will improve if we keep track of the number of hours used for testing each day. The data used for prediction in this example was collected from error reports so, we do not know exactly how effective the testing was in different periods. We are, however, convinced that if data are collected directly for the model, we will get very good estimates of the times concerned. Before we can state this definitely we have to evaluate the model carefully on some other projects.

The model seems to have a high potential as a tool for prediction of the time until a number of errors has been found, during function testing.

7. CONCLUSIONS

We have developed a model to be used for prediction of software error occurrences during testing done in parallel, e.g. function testing. It is essential to try to understand the stochastic behaviour of how software errors occur, in order to obtain an efficient, reliable, maintainable and manageable software product.

This can only be done if we make a survey of software projects and find the critical parts, the main points, and get an overall view of the behaviour of the underlying processes. This survey has already been done for hardware, but it is also necessary to do it for software. Our current systems are continually getting larger and more complex. If we do not want to end up with a software product that is out of control, it is necessary to adopt or adapt the model presented in this paper or develop a model similar to this one.

The process of developing and maintaining software will soon be so complicated that we are bound to lose control of it if we do not establish a mathematical and scientific foundation for Software Engineering. A more scientific view would help to make Software Engineering into an industrial process instead of looking at it as an art. We are convinced that it is essential to use models and metrics for parameters such as complexity, reliability and quality in the future, in order to be able to manage projects and develop high quality products.

8. ACKNOWLEDGEMENT

This project is supported by Telelogic AB and the Swedish Telecommunication Administration, Sweden.

9. REFERENCES

1. Boehm, B., 1981, "Software Engineering Economics", Prentice-Hall Inc., Englewood Cliffs, USA.

2. Wohlin, C., 1985, "Software Testing", Lund Institute of Technology, Lund, Sweden, (in Swedish).

3. Goel, A., 1983, "A Guidebook for Software Reliability Assessment", Syracuse University, Syracuse, USA.

4. Lennselius, B., 1986, "Software Complexity and its Impact on Different Software Handling Processes", Proc. IEE, 259, 148-153.

5. Jelinski, Z., and Moranda, P., 1973, Statistical Computer Performance Evaluation, Academic Press, 465-484.

6. Goel, A., and Okumoto, K., 1979, IEEE Trans. on Reliability, Vol. R-28, No 3, 206-211.

7. Mills, H.D., 1972, "On the Statistical Validation of Computer Programs", IBM Federal Systems Division, Gaithersburg, MD, USA, Report 72-6015.

8. Kleinrock, L., 1976, "Queueing Systems, Vol 1. Theory", John Wiley and Sons, New York, USA.

9. Wohlin, C., 1986, "A Comparison Between Two Software Reliability Models", Technical Report, Lund Institute of Technology, Lund, Sweden.

10. Vrana, C., and Wallander, A., 1983, "S/W Quality and Complexity - Different Aspects and Measurement Results", Proc. IEE, 223, 121 -127.

11. Wohlin, C., and Vrana, C., 1986, "A Quality Constraint Model to be Used During the Test Phase of the Software Lifecycle", Proc. IEE, 259, 136-141.

12. Feller, W., 1957, "An Introduction to Probability Theory and its Applications", John Wiley and Sons, New York, USA.

13. Karlsson, J., and Stavenow, B., 1980, "Methods for Numerical Inversion of Laplace- and z-transform", Technical Report, Lund Institute of Technology, Lund, Sweden.

14. DeMarco, T., 1982, "Controlling Software Projects", Yourdon Press, New York, USA.