Prediction of Software Failures Based on Systematic Testing*

Michael Grottke
Chair of Statistics and Econometrics
University of Erlangen-Nuremberg
Lange Gasse 20
D-90403 Nuernberg
Germany

Klaudia Dussa-Zieger
imbus AG
Kleinseebacher Str. 9
D-91096 Moehrendorf
Germany

Contact: Michael.Grottke@wiso.uni-erlangen.de

Abstract

Software can be tested in several ways: either according to the operational profile of the user, or by following a systematic testing approach. If the latter is chosen, then it becomes difficult to estimate the number of failures in the software, or its reliability. In this paper, we present a software reliability model, the Rivers-Vouk model, which specifically addresses the issues of systematic testing. Using an intuitive model framework its assumptions are compared to those of a model designed for operational testing. The Rivers-Vouk model has been applied to a test project at imbus AG and the results of the failure prediction are discussed. In the conclusion the PETS project is briefly reviewed whose aim it is to incorporate data on the software process maturity into software reliability models.

1 Software testing and failure models

No piece of software, independent of its size and complexity, is free of faults. As software is written by humans, errors will always occur. Due to a wrong or incomplete specification, large problem complexity, lack of time, etc. mistakes are made, and when this happens whilst developing software this is known as an error. The result of a human error being made is a software fault[1], i.e. an incorrect piece of software. When the faulty software is executed, it can exhibit an unexpected behavior, produce an incorrect result. This is known as a failure. Software failures are a serious matter. In a growing and increasingly networked information society, in which the computer has conquered almost every aspect of our lives, a software failure may result in an threat to life or assets. In the Therac-25 accident, for example, several patients lost their lives due to a radiation overdose. Statistics on software failures published in literature clearly document the risks [16].

In this situation it is of vital interest that we thoroughly test critical software and are able to predict the number of failures experienced in the remaining test time or the number of faults in the software after release. With this goal in mind numerous statistical models have been developed. The basic approach is to model past failure data to predict future behavior. These models are typically based on failure data such as the number of failures experienced in specified time intervals, or the time elapsed between two consecutive failures.

Reflecting the definition of a failure, it becomes clear that the occurrence of a failure depends on the characteristics of the software and on the way in which the software is used. One possible way to test software is according to the typical behavior of the user, which is also called the operational profile. Testing according to the operational profile means that the operations of the system under test have to be determined together with their execution probabilities. Based on these probabilities the number of test cases is allocated to the different program functions. The test cases should then be executed in a random order, i.e. different parts of the software should not be tested in strict succession. Operational testing[2] has two advantages. Since the operations that will be used frequently by the users are tested more thoroughly, more failures will occur in these parts of the software during testing and consequently these parts will be more stable afterwards (assuming a

*This work was supported by the European Community in the framework of the specific programme for research, technological development and demonstration on a user friendly society (1998-2002), the "IST Programme", Contract No. IST-1999-55017. The authors are solely responsible for this paper. It does not represent the opinion of the Community, and the Community is not responsible for any use that might be made of data appearing herein.
This article was published in: Electronic Proc. Ninth European Conference on Software Testing Analysis and Review (EuroSTAR), Stockholm, 2001

perfect removal of the faults) [9]. If the usage during testing does not differ from usage after release of the software, then the failure behavior is close to the one the user would also have experienced. Based on the failure data the reliability, i.e. the probability of a failure-free operation of the software for a specified interval of time, can be calculated [2].

In the vast majority of industrial organizations testing is done in a systematic manner [3]. Based on the functional specification of the application under test or its program code, the test cases are developed. The tester pays attention to all aspects of the software. Given the specific area of application the complete test of all software requirements may even be mandatory due to legal requirements, e.g. FDA asks for a complete requirements tracing for medical software. Following this testing strategy the goal is to provoke as many failures as possible, to correct the underlying faults and thereby increase the software reliability. However, it is not possible to assess the reliability based on the collected failure data since the aspect of operational testing is neglected. It is considered more important to test the entire functionality of the application.

To execute tests in a systematic and efficient manner, test cases are often specified following black box design techniques. Black box testing views the application under test as a black box, i.e. the program code is hidden and only the functional specification of the application, the expected behavior at the interface to the user, is known. Classical black box techniques are equivalence partitioning and boundary value analysis [10, 16]. Using equivalence partition testing the possible values for input and output parameters of the application are divided into non-overlapping classes in such a way that the behavior of the software is assumed to be the same for all members of one class. Only one representative of each class has to be tested. In most cases the partitions are derived from the functional specification of the software. Test designers may also make use of their knowledge about typical errors of programmers and of their intuition for creating the partitions. Boundary value analysis identifies the boundaries between equivalence classes and treats them as additional partitions. Experience has shown that the boundary values tend to be error-prone, and therefore additional test cases are designed.

Besides the black box techniques also white box techniques can be applied for the derivation of test cases [16]. Here the internals of the application, i.e. the program code, are given, whereas the functional specification is not regarded. In white box testing a criterion referring to code constructs (e.g. lines or branches) is defined for test case derivation. Based on the chosen criterion the test cases are generated with the intention to cover as many constructs of the code as possible. Occasionally black box testing and white box testing are combined. Here, a base set of black box tests is defined. While they are executed, their structural coverage is measured. Afterwards additional tests are defined based on the knowledge of the program code to further enhance the coverage.

To increase the speed of test case execution, test cases are often ordered in a clever way such that preconditions for the following test case are established by the prior test. Thus, the effort to set up specific test conditions and navigate through the application is minimized.

Testing in a systematic manner has several advantages. Systematic testing avoids redundancies in the test cases. Navigation through the application is minimized. Critical, but rarely used functions are tested since the complete functionality of the application is tested. It allows to start with testing when no operational profile is available or can be determined, e.g. for new applications where no historical data is available. Looking at the characteristics of operational testing and systematic testing, it is obvious that both approaches differ substantially.

Classical software reliability models assume that the failure data was collected during testing following the operational profile. Another assumption of these models is that the faults are corrected instantaneously and perfectly. As a consequence the mean value function μ, which represents the expected number of failures experienced, has a decreasing slope starting from a certain point of usage. The reliability as it is experienced by the user increases. Therefore, the models are called software reliability growth models. Per se these models are not appropriate to model the failure behavior during systematic testing. Further analysis and interpretation of the model properties is necessary to adapt them to different testing situations.

2 Derivation and discussion of the Rivers-Vouk model

As shown by Grottke [7], several finite failure category models fit into a common framework. According to this framework, the functional form of the number of failures experienced in calendar time is determined by the following relationships:

1. The allocation of testing effort, i.e. time spent on executing the application under test, to calendar time
2. The development of the number of test cases executed as a function of cumulative testing effort

3. The coverage of code constructs attained through test case execution

4. The relationship between structural coverage and the number of failures experienced

Not all relationships have to be included in a model. Indeed, most software reliability growth models start out with a measure of testing effort (like CPU time used by the application under test, the so-called execution time), not with calendar time. Moreover, many models directly express the development of the cumulative number of failure occurrences as a function of testing effort and do not explicitly specify the intermediate relationships. However, these relationships may help to derive and in this way explain the overall functional form of different software reliability growth models.

How structural coverage changes with the execution of additional test cases will depend on the number of code constructs sensitized by the test cases, and on the extent of redundancy in selecting these constructs.

A possible setup is the one of operational testing with a homogenous operational profile, in which all non-overlapping and equally-sized operations have the same occurrence probability: Per test case, p of the G code constructs a piece of software consists of are exercised on average. The p constructs are always sampled from the entire population, i.e. constructs may be tested over and over again. Piwowarski et al. [12] have shown that these assumptions lead to an exponential form of the expected structural coverage, C, as a function of the number of test cases executed, i:

$$E(C(i)) = 1 - \exp\left(-\frac{p}{G}i\right) \tag{1}$$

The continuous approximation of this relationship is depicted by the solid line in figure 1 for $\frac{p}{G} = 0.002$. Obviously, even though the number of constructs exercised per test case stays constant, the number of the ones sensitized for the first time decreases due to the redundant execution of constructs already tested before.

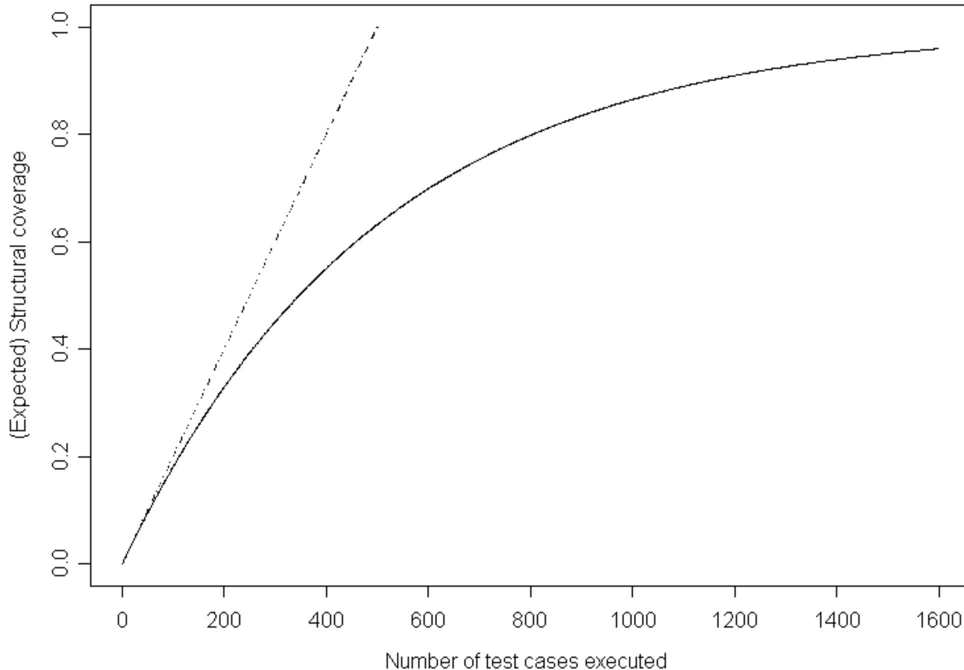

Figure 1: *(Expected) Structural coverage attained with replacement (—) or elimination (·-) of all tested constructs, with* $\frac{p}{G} = 0.002$

If we are interested in the formulation of the number of failures experienced in terms of testing effort, then we also have to specify the relationship between testing effort and the number of test cases executed and the development of the number of failure occurrences in dependence of structural coverage. Under the assumption that for each test case the same amount of testing effort t_t is necessary, the number of executed test cases is proportional to the testing effort spent. Likewise, the second relationship is that of proportionality if the N faults expected to be located in the software code at the beginning of testing[3] are uniformly distributed over

the G constructs, if a fault causes a failure the first time the construct in which the fault is located is sensitized, and if each fault is removed instantaneously and perfectly as soon as it has caused a failure.

With these suppositions, the shape of the mean value function in terms of testing effort t is an exponential one, like for structural coverage in dependence of the number of test cases:

$$\mu(t) = E(M(t)) = N\left[1 - \exp\left(-\frac{p}{G \cdot t_t} t\right)\right] \quad (2)$$

This is the form of the widely-used model by Goel and Okumoto [5]. As our derivation shows, its ever-decreasing slope (cf. figure 2) - i.e., the testers' declining effectiveness in detecting faults by "producing" software failures - can be explained by the inefficient sampling in the framework of operational testing.

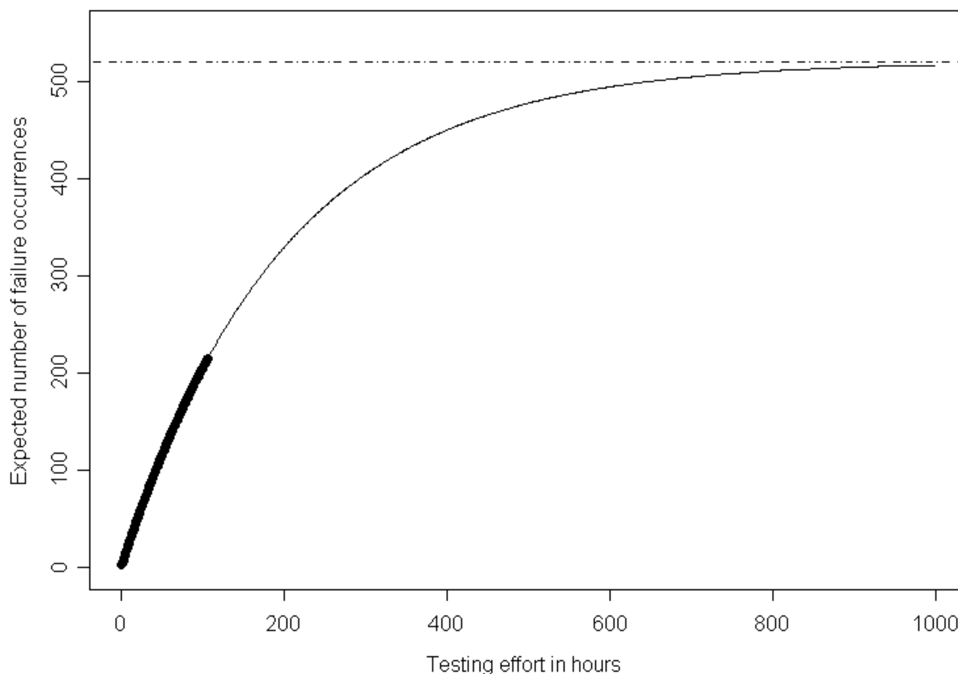

Figure 2: *Mean value function of the Goel-Okumoto model with N=520 and* $\frac{p}{G \cdot t_t} = 0.005$

Practical application of the model has often shown that it is too optimistic: Since the parameter N stands for the total number of faults in the software at the beginning of testing, and since each fault is assumedly corrected as soon as it has caused a failure, the estimated value $\hat{N}$ calculated at some point during testing is also an estimate for the maximum total number of failure occurrences until the end of testing. However, the cumulative number of failures experienced often exceeds this estimate after some additional testing [1, p. 15]. Our discussion suggests that this behavior may be due to a disparity between model assumptions and the real world: Testers actually do not sample the code constructs with perfect replacement, because they try to reduce at least some redundancy in testing. Therefore, the decrease in the slope of the structural coverage function (and of the mean value function as well) is not as large as presumed by the Goel-Okumoto model. As a consequence, the estimated number of failures to be experienced until the end of testing is too small.

It can be expected that software reliability growth models based on the perfect-replacement-of-constructs assumption are especially inappropriate for analysis of failure data collected during systematic testing. A model designed for this testing strategy has to specify a different relationship between the number of test cases executed and structural coverage attained.

One of the few models for systematic testing was developed by Rivers and Vouk [13, 15], building on earlier work [14, 17]. They consider test data which is grouped in non-overlapping stages of testing. For each stage j, which consists of one or more test cases, the number of test cases executed (Δi_j) or code constructs exercised (Δq_j) at this stage has to be available in addition to the number of failure occurrences (Δm_j). A central assumption of their model is that code constructs which have already been tested are not tested a second time. Since Rivers and Vouk equivalently use code coverage metrics and the number of test cases executed as input

values of their model, they obviously suppose proportionality: the coverage achieved by the execution of k test cases is k times the coverage attained by one test case. This means that the test cases are equally-sized with respect to the number of code constructs sensitized per test case. For example, if a software code consists of $G = 4000$ constructs, 8 of which are newly exercised by each test case, then per test case 0.2 per cent additional structural coverage is attained; after 500 test cases all constructs have been covered. The dotted line in figure 1 depicts this relationship and can be compared to the result of the operational testing setup.

Most of the few models explicitly specifying how the number of failures experienced develops in dependence of structural coverage - like the coverage model by Piwowarski et al. [12] discussed above or the Enhanced Non-homogeneous Poisson Process framework by Gokhale et al. [6] - presume proportionality. However, this assumption seems to neglect important aspects: First of all, even when all G constructs of a software code have been tested not all faults have necessarily caused a failure. There are faults which are not detectable under a given testing strategy. Therefore, good testing does not only increase structural coverage, but also does it "the right way", i.e. by covering constructs that are thought to be error-prone. Furthermore, more than one construct may lead to the detection of the same fault. This increases the visibility of such faults and the ease with which they can be detected. If faults with high activation probabilities tend to be found earlier, as might be expected, the average visibility of the faults will decrease as testing proceeds, because especially hard-to-detect faults remain in the software. On the other hand, the growing experience of testers may increase their insight into where faults are hidden and into their ability to run test cases which activate these faults.

To account for such phenomena, Rivers and Vouk introduce a factor g_j, which relates the number of faults remaining in the software at the beginning of the jth stage of testing to the number of constructs whose execution causes a failure: If the number of failures experienced and faults corrected during the first $(j-1)$ stages is m_{j-1}, then the number of faults visible at the jth stage is therefore $g_j\,(N - m_{j-1})$. Since it is also related to the testers' performance, Rivers and Vouk refer to g_j as the "testing efficiency". The probability of experiencing Δm_j failures by testing Δq_j constructs at this stage is then the number of ways in which exactly Δm_j constructs can be chosen out of the $g_j\,[N - m_{j-1}]$ faulty ones divided by all possible combinations of sampling Δq_j constructs out of the $G - q_{j-1}$ constructs which have not been exercised (and removed) before. This line of thoughts leads to the following hypergeometric probability function:

$$P(\Delta M_j = \Delta m_j \mid m_{j-1}) = \frac{\binom{g_j(N - m_{j-1})}{\Delta m_j}\binom{[G - q_{j-1}] - [g_j(N - m_{j-1})]}{\Delta q_j - \Delta m_j}}{\binom{G - q_{j-1}}{\Delta q_j}}, \tag{3}$$

for Δm_j taking integer values smaller than or equal to Δq_j.[4]

From this probability function, the number of failures we expect to experience at the jth stage if we know that m_{j-1} failures have occurred before that stage, can be derived by applying a simple rule [11, p. 177]:

$$E(\Delta M_j \mid m_{j-1}) = g_j(N - m_{j-1})\frac{\Delta q_j}{G - q_{j-1}} = g_j(N - m_{j-1})\frac{\Delta c_j}{1 - c_{j-1}}. \tag{4}$$

$\Delta c_j = \frac{\Delta q_j}{G}$ is the structural coverage gained at stage j, and $c_{j-1} = \frac{q_{j-1}}{G}$ is the coverage at the beginning of that stage.

With equation (4) we can only calculate the expected number of failures at a certain stage if all earlier stages have already taken place. If we want to make long-term predictions, we have to use an expression which does not depend on this knowledge, namely the unconditional expected value of ΔM_j:

$$E(\Delta M_j) = g_j[N - E(M_{j-1})]\frac{\Delta c_j}{1 - c_{j-1}}. \tag{5}$$

In equation (5), the actual number of failure occurrences up to the $(j-1)^{th}$ stage, m_{j-1}, has been replaced by its expected value, $E(M_{j-1})$. This equation can be used as a starting-point for deriving the mean value function $\mu(c)$, i.e. the expected cumulative number of failures experienced when $100 \cdot c$ % coverage has been achieved. For this end we have to conceptualize infinitely small stages which lead to an increase in coverage of almost zero. Consequently, the number of failure occurrences after the jth stage may be very close to the one after the $(j-1)^{th}$ stage, and equation (5) can be continuously approximated by

$$d\mu(c) = g(c)[N - \mu(c)]\frac{dc}{1 - c}, \tag{6}$$

where dc and $d\mu$ indicate the infinitely small increases in structural coverage and in the expected number of failures, respectively. The mean value function is then [13, pp. 38 - 39]

$$\mu(c) = N - (N - \mu_{min}) \exp\left(- \int_{c_{min}}^{c} \frac{g(\varsigma)}{1-\varsigma} d\varsigma \right). \tag{7}$$

c_{min} and μ_{min} denote the minimum meaningful coverage and the expected number of failures observed for this coverage. They are solely introduced for stabilization purposes and do not change the shape of the mean value function, but move its origin in a cartesian coordinate system from (0, 0) to the point (c_{min}, μ_{min}).

However, the shape of the mean value function depends on what form the testing efficiency (TE) takes. Rivers considers three different cases [13, pp. 40 - 42]:

1. Assuming a constant testing efficiency, i.e. setting $g(c) = \alpha$ with $\alpha > 0$, yields

$$\mu(c) = N - (N - \mu_{min}) \left(\frac{1-c}{1-c_{min}} \right)^{\alpha}. \tag{8}$$

   A property of this model is that at full coverage ($c = 1$) all N faults in the software code are expected to have caused failures. For $g(c) = \alpha = 1$, when each fault is visible through exactly one construct, the number of failures experienced is proportional to coverage, and the software reliability growth model is a straight line. For $g(c) = \alpha > 1$, the faults are highly visible and therefore will be detected early. This means that the model starts out with a high slope which continuously decreases, because less faults remain in the software at higher coverage levels. Just the opposite holds true for $0 < \alpha < 1$. This behavior in connection with a *constant* testing efficiency is clearly counterintuitive.

2. Using the linear testing efficiency $g(c) = \alpha(1-c)$, which decreases from $\alpha > 0$ at the beginning of testing to zero at full coverage, results in

$$\mu(c) = N - (N - \mu_{min}) \exp\left(-\alpha(c - c_{min})\right). \tag{9}$$

   Note that the shape of this function is exactly the one of the Goel-Okumoto model (cf. equation (2)). However, its interpretation differs from the one by Piwowarski et al. described at the beginning of this section. While in their derivation inefficient sampling of code constructs leads to the exponential form of coverage growth, here coverage develops proportionally to the number of test cases, and the decreasing efficiency in detecting faults is the reason for the shape of the overall function. Therefore, we see that the Goel-Okumoto model is not necessarily connected to operational testing. However, it always assumes inefficiency in one of the relationships.

   Another important difference is that the independent variable in equation (9), coverage c, can only take values between zero and one, unlike testing effort in equation (2), which at least conceptually can reach infinity. Since $\exp\left(-\alpha(1 - c_{min})\right)$ is always larger than zero (as long as α is not extremely large and c_{min} is not equal to one, which both would lead to a useless model anyway), according to the model a number of faults will never be detected.

3. If the testing efficiency is assumed to take the form $g(c) = \alpha\gamma c^{\gamma-1}(1-c)$ with $\alpha > 0$ and $\gamma > 0$, which is a generalization of the linear testing efficiency function, then the following mean value function is obtained:

$$\mu(c) = N - (N - \mu_{min}) \exp\left(-\alpha(c^{\gamma} - c_{min}^{\gamma})\right). \tag{10}$$

   This equation corresponds to the mean value function of a software reliability model referred to as the "Goel generalized nonhomogeneous Poisson process model" [4], the "generalized Goel-Okumoto model" [6] or the "Weibull model" [2] in literature.

   For $\gamma > 1$, the testing efficiency increases in the early stages of testing until it reaches a maximum value and then decreases to zero. Therefore, this model is referred to as the unimodal TE model. Due to the increasing/decreasing efficiency in detecting those code constructs where faults are located, the mean value function is S-shaped. The linear TE model is obtained for $\gamma = 1$. In this case as well as for $0 < \gamma < 1$ the testing efficiency is strictly decreasing.

   Like in the linear TE model, several of the N faults in the software code will not have been detected even when full coverage will be attained.

For a real data set of coverage levels and the respective cumulative numbers of failure occurrences, the parameter values of (8) to (10) - N, α and β - can be estimated by searching for those values that minimize the sum of squared errors (SSE) of the fitted model.

Due to the implicitly assumed proportionality between the number of test cases executed and structural coverage c, the three forms of the Rivers-Vouk model can also be expressed in terms of test case coverage b, the number of test cases executed divided by the total number of test cases planned, by substituting b for c. If the number of failure occurrences is not only known for sets of test cases but individually per test case, the minimum meaningful test case coverage b_{min} and - consequently - μ_{min} should be set to zero.

Rivers argues that a non-increasing testing efficiency being appropriate for modelling the number of failure occurrences indicates a lack of learning within the testing project. Testers are not able to or not allowed to run test cases departing from the test specification and therefore cannot use the experience made to improve their effectiveness [13].

Two remarks to this assertion seem to be necessary:

1. We should not forget that the so-called "testing efficiency" $g(c)$ is by no means the only aspect concerning the testers' performance. Its influence is restricted to the relationship between structural coverage and the number of failures experienced. However, metrics of effectiveness and efficiency can also be defined for the second and third relationship listed at the beginning of this section (while the first one is more a matter of allocation of resources). For example, the average time needed per test case execution depends on whether the test cases have been ordered in some convenient way minimizing navigation between different parts of the application under test. Moreover, as the testers gain experience in using the tested software, they will probably be able to execute more test cases in a certain amount of time. Therefore, the term "testing efficiency" sounds more comprehensive than the scope of $g(c)$ in fact is. Judgements about whether any learning takes place during the testing process cannot be solely based on the functional form of $g(c)$.

2. As for the structural coverage attained through test case execution, the Rivers-Vouk model implicitly supposes proportionality, which is a strong assumption. In reality, perfect efficiency in sampling code constructs will hardly be achieved, especially if test cases are specified using a black box approach. If a model assuming a constant testing efficiency in addition to non-redundant sensitization of code constructs is appropriate for modelling the collected failure data, this seems to indicate a good overall quality of the test specification.

3 Application of the model

The three different forms of the Rivers-Vouk model have been used for the analysis of a testing project at imbus. The application under test was the first release of a large customized software for a public authority.

How to provide for the necessary data was a first issue. For several years, during each project information about the failures themselves, like a detailed description, the day of occurrence and the severity had already been collected in a database. However, a measure of testing effort, or a record unambiguously stating the test case whose execution lead to the failure were not ready at hand. Therefore, to gather the data appropriately, additional effort had to be invested. *ProDok*, a home-grown test plan management tool of imbus, was augmented with functions for logging the time intervals in which a test case is executed, and the failure occurrences. This new part of *ProDok* is called *Teddi* (*te*st *d*ata *d*etermination and *i*nterpretation tool). From the raw data collected, for each test case the time spent on it and the number of failures experienced can be calculated among other metrics.

The determination of the order in which the test cases are executed poses another practical problem, because the test cases are often not tested in strict succession. For example, at the end of a test cycle, a tester might come back to a test case almost entirely executed before and spend some few additional minutes on it. Since the test cases are atomic units, considering the test case finished at this later point has the consequence of counting *all* failures that occurred for the test case as if they were experienced at the end of the test cycle. A pragmatic solution has been implemented: To each testing interval collected for a test case a factor between zero and one is assigned according to its share at the total testing effort spent on this test case. The weighted sum of the end points of the intervals then yields the "average time of completion" of the test case. In the analysis, the test cases are chronologically ordered according to these values.

As for the project under consideration, after about 76 per cent of the planned test cases had been executed, 190 failures caused by different faults had been reported by the testers. Figure 3 shows $M(b)$, the number of

failure occurrences as a function of the fraction of test cases executed.

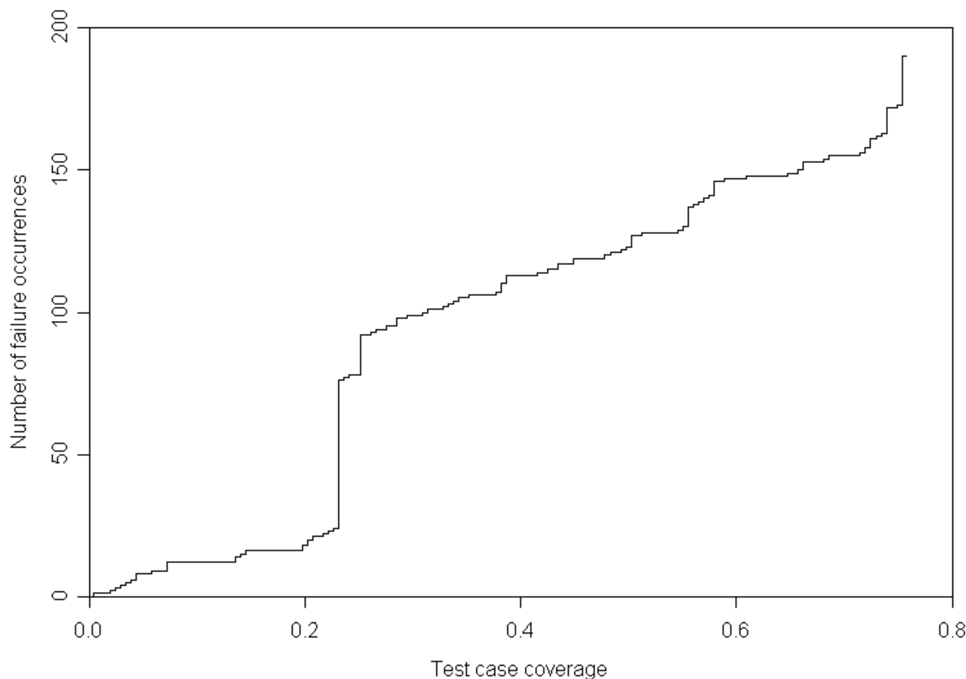

Figure 3: *Test case coverage and number of failures experienced*

While the number of failure experienced per test case is moderate in most cases, some jumps in the diagram caused by test case executions resulting in a large number of failure occurrences are clearly visible, for example at about 25 per cent of test case coverage. A reason for this might simply be that highly defective code was tested by these test cases. However, it can also hint at extensive test cases sensitizing much more constructs than the average test case does. Rather than a high fault density of the code, the sheer testing effort spent would then be responsible for the effect noticed. Since the Rivers-Vouk model implicitly assumes that the test cases are equally-sized, it is important to identify severe deviations of real-world conditions from this proposition and take corrective measures.

Plotting for each test case the time spent on its execution against the number of failure occurrences (figure 4) indeed reveals that for those test cases causing the jumps in figure 3 the testing effort in terms of time is clearly above average.

As a remedy, we split each of the four test cases with a testing effort of more than 300 minutes into several medium-sized ones, randomly allocating the failures of the original test case. Since we did not want this artificial increase in the number of executed test cases to raise the value of test case coverage, the number of planned test cases was also multiplied by the factor

$$\frac{\text{Number of executed test cases in the adjusted data set}}{\text{Number of executed test cases in the original data set}}.$$

The resulting step function is depicted in figure 5.

The three forms of the Rivers-Vouk model, expressed in terms of test case coverage b, were fitted to the adjusted data set. b_{min} and μ_{min} were both set to zero. The resulting estimates of N, α and - where it applies - β as well as the SSE and the estimated number of failures observed at full test case coverage are listed in table 1.

Model fitted	$\hat{N}$	$\hat{\alpha}$	$\hat{\beta}$	SSE	$\hat{\mu}(1.0)$
Constant TE model	202.35	1.332	-	15000	202.35
Linear TE model	518.37	0.535	-	14732	214.63
Unimodal TE model	268.80	1.376	1.166	13925	200.93

Table 1: *Results of fitting the different forms of the Rivers-Vouk model to the data set*

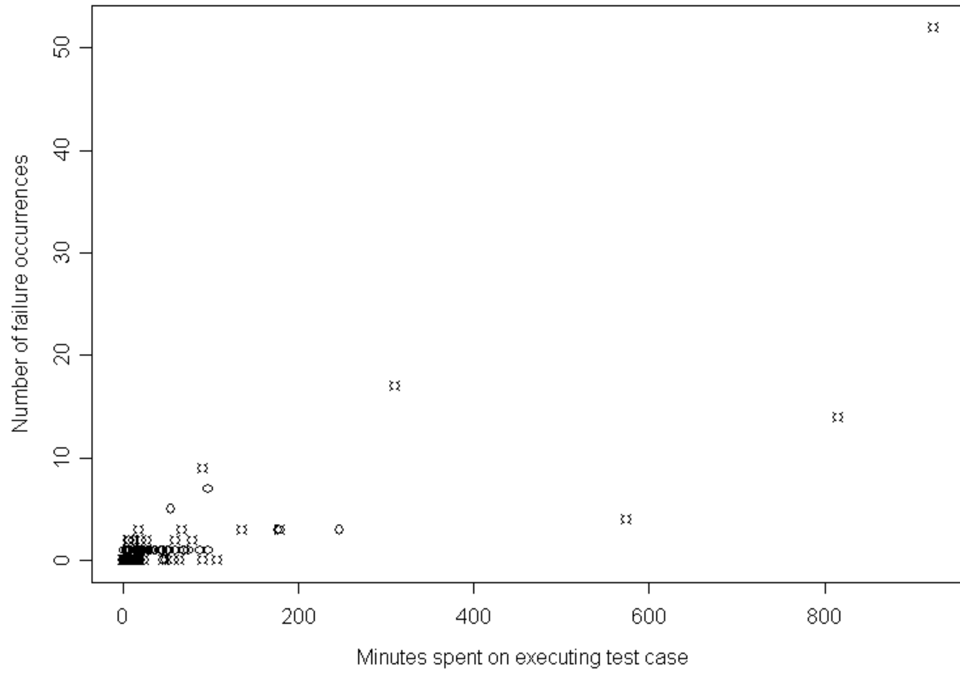

Figure 4: *Minutes spent on testing and number of failure occurrences for each test case*

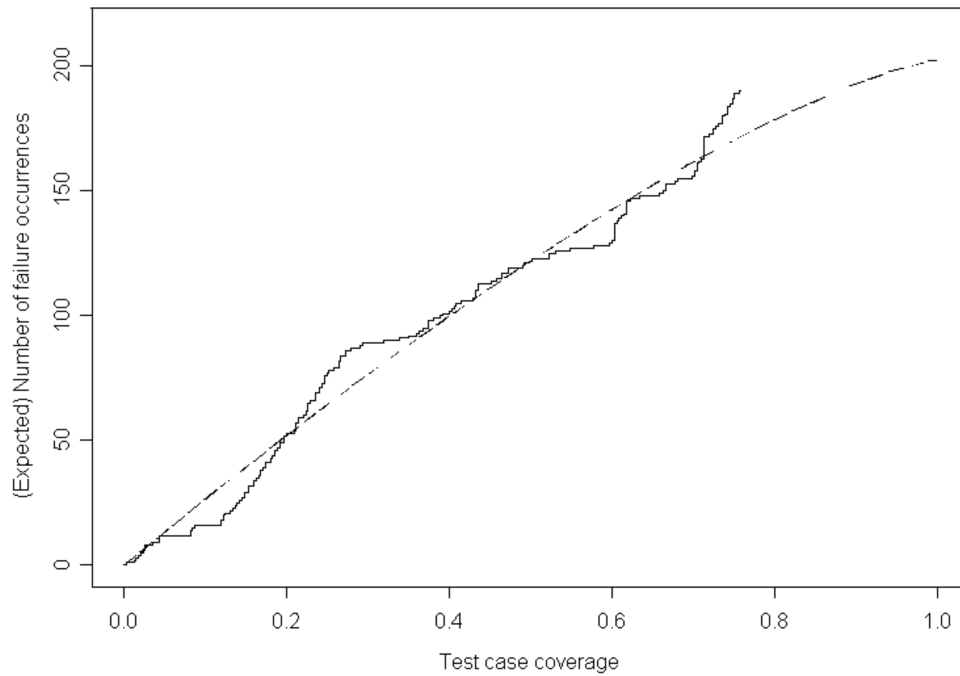

Figure 5: *Test case coverage and number of failures experienced (⌜), and mean value function of the fitted constant TE model (— —)*

In figure 5 the mean value function of the fitted constant TE model is shown, and in figure 6 the mean value functions of the fitted linear TE model and the fitted unimodal TE model are depicted.

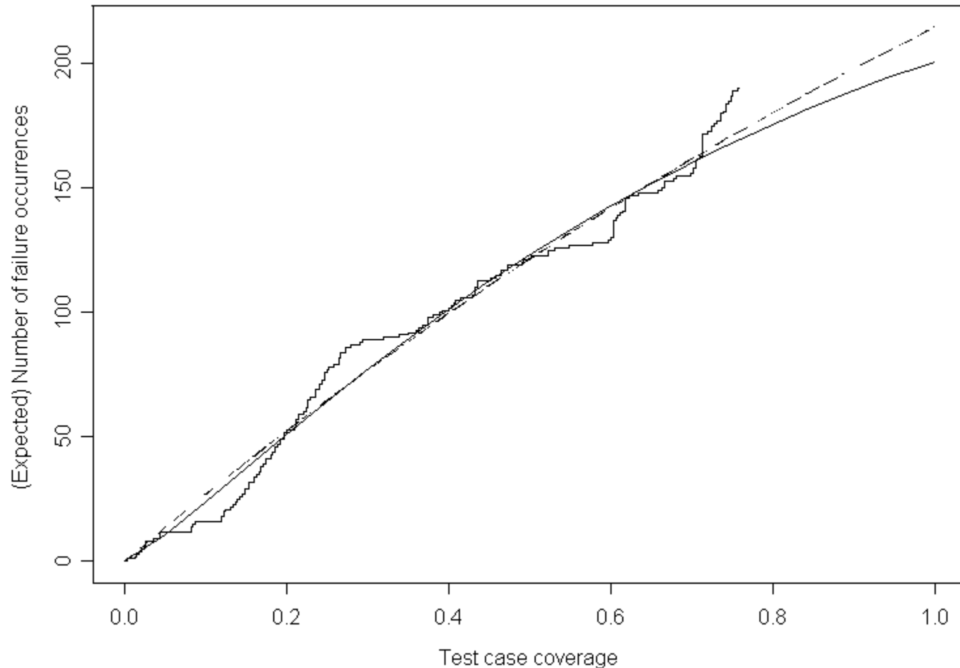

Figure 6: *Test case coverage and number of failures experienced (⌜), and mean value functions of the fitted linear TE model (— —) and the fitted unimodal TE model (—)*

Although the relationship between the number of failure occurrences and test case coverage is almost linear in the adjusted data set, which seems to hint at a constant testing efficiency of about one, the linear TE model and the unimodal TE model yield a better fit than the constant TE model in terms of the SSE criterion. Since the unimodal TE model is a generalization of the linear TE model with an additional parameter, the fit of the former necessarily has to be at least as good as the one of the latter. The fitted mean value functions of all models are quite similar and predict between eleven and twenty-four additional failures to be experienced until all of the planned test cases have been executed.

Looking at the estimated number of faults inherent in the software at the beginning of testing, confirms the suspicions we already have in regard to the excellent fit of all three models: According to the unimodal TE model about 68 faults will not have been detected at the end of testing, and according to the linear TE model this number is even 303!

Since the linear TE model has the shape of the Goel-Okumoto model, figure 2 helps us to understand the reason for both its good fit and its unconvincing estimates: That only about 41 per cent of the faults will have caused failures after the execution of all test cases basically means that we will only experience the first part of the mean value function (drawn as a thick line), which is almost linear. Therefore, although the linear TE model is designed for a situation in which the number of failures experienced per test case decreases as testing continues, it can fit linear relationships well if only the estimated number of initial faults, $\hat{N}$, is large enough.[5] Similar arguments apply to the more general unimodal TE model.

As this short analysis of the data set has shown, even if the estimated mean value function of a model seems reasonable in the region from zero coverage to full coverage, care must be given to the interpretation of model parameters. If model assumptions (like decreasing testing efficiency) are clearly not in accordance with real-world conditions, the model should not be used.

4 Conclusions

This research was done in the course of the PETS project (Prediction of software Error rates based on Test and Software maturity results) funded by the European Commission. One of the goals of this project is to develop an enhanced software reliability growth model which makes use of information about the maturity of the software development process and the software test process. Since all of the companies participating at the project employ some sort of systematic testing regime, it was important to investigate in which way classical software reliability growth models rely on the fact that the failure data is collected during operational testing.

Using the model framework developed in [7] as a starting point for the derivation and interpretation of the Rivers-Vouk model revealed two merits of this model: First of all, it implicitly assumes perfect efficiency in sampling code constructs, which is one of the goals of systematic testing strategies. Moreover, unlike other models specifying the relationship between structural coverage and the number of failure occurrences, the Rivers-Vouk model takes into account that the latter is not (necessarily) proportional to the former.

Of course, the proposition that all tested code constructs are eliminated from further consideration marks a limiting case which is not achieved in real world - just like the setup in the model by Piwowarski et al. also discussed in this paper. While the specification of different testing efficiency functions lays the foundation for choosing from a variety of models the one which yields the best fit and for drawing conclusion about the testing process, application of the three forms of the Rivers-Vouk model has shown that the estimated model closest to the actual data is not necessarily the one whose assumptions correspond best to reality. Therefore, caution must be given to the interpretation of model parameters and of the estimated testing efficiency. Finally, it should be understood that the scope of the concept of "testing efficiency" in the Rivers-Vouk model is more limited than the name suggests.

Notes

[1]Faults are also named defects, or bugs.

[2]Operational testing is also called representative testing, or statistical usage testing.

[3]More precisely, the number of inherent faults is a random variable following a Poisson distribution with expected value N.

[4]Problems of this formulation have been discussed by Grottke et al. [8].

[5]An explanation based not on the shape of the mean value function but on the estimated empirical testing efficiency is given in [8].

References

[1] Denton, J. A.: *Accurate Software Reliability Estimation*, Thesis, Colorado State University, 1999, URL = http://www.cs.colostate.edu/~denton/jd_thesis.pdf (site visited 2001-05-31)

[2] Farr, W.: *Software Reliability Modeling Survey*, in: Lyu, M. R. (ed.): *Handbook of Software Reliability Engineering*, New York, San Francisico, et al., 1996, pp. 71 - 117

[3] Frankl, P. G.; Hamlet, R. G.; Littlewood, B.; Strigini, L.: *Evaluating Testing Methods by Delivered Reliability*, IEEE Trans. Software Eng. 24 (1998), pp. 587 -601

[4] Goel, A. L.: *Software Reliability Models: Assumptions, Limitations, and Applicability*, IEEE Trans. Software Eng. 11 (1985), pp. 1411 - 1423

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

[6] Gokhale, S. S.; Philip, T.; Marinos, P. N.; Trivedi, K. S.: *Unification of Finite Failure Non-Homogeneous Poisson Process Models through Test Coverage*, Technical Report 96-36, Center for Advanced Computing and Communication, Department of Electrical and Computer Engineering, Duke University, 1996, URL = ftp://ftp.eos.ncsu.edu/pub/ccsp/papers/ppr9636.PS (site visited 2001-05-31)

[7] Grottke, M.: *Software Reliability Model Study*, Deliverable A.2 of project PETS (Prediction of software Error rates based on Test and Software maturity results), IST-1999-55017, 2001

[8] Grottke, M.; Dussa-Zieger, K.: *Systematic vs. Operational Testing: The Necessity for Different Failure Models*, to appear in: Proc. Fifth Conference on Quality Engineering in Software Technology, Nuremberg, 2001

[9] Musa, J. D.: *Operational Profiles in Software-Reliability Engineering*, IEEE Software, March 1993, pp. 14 - 32

[10] Myers, G. J.: *Methodisches Testen von Programmen*, 6th edition, München, Wien, et al., 1999

[11] Newbold, P.: *Statistics for Business and Economics*, 3rd edition, Englewood Cliffs, 1991

[12] Piwowarski, P.; Ohba, M.; Caruso, J.: *Coverage Measurement Experience During Function Test*, Proc. Fifteenth International IEEE Conference on Software Engineering (ICSE), 1993, pp. 287 - 301

[13] Rivers, A. T.: *Modeling Software Reliability During Non-Operational Testing*, Ph.D. thesis, North Carolina State University, 1998, URL = http://renoir.csc.ncsu.edu/Faculty/Vouk/Papers/Rivers/Thesis/Rivers.Thesis.pdf.zip (site visited 2001-05-31)

[14] Rivers, A. T.; Vouk, M. A.: *An Empirical Evaluation of Testing Efficiency during Non-Operational Testing*, Proc. Fourth Software Engineering Research Forum, Boca Raton, 1995, pp. 111 - 120, URL = http://renoir.csc.ncsu.edu/Faculty/Vouk/Papers/SERF96.ps (site visited 2001-05-31)

[15] Rivers, A. T.; Vouk, M. A.: *Resource-Constrained Non-Operational Testing of Software*, Proc. Ninth International Symposium on Software Reliability Engineering, Paderborn, 1998, pp. 154 - 163

[16] Thaller, G. E.: *Software-Test: Verifikation und Validation*, Hannover, 2000

[17] Vouk, M. A.: *Using Reliability Models During Testing With Non-Operational Profiles*, Computer Science Department, North Carolina State University, 1992, URL = http://renoir.csc.ncsu.edu/Faculty/Vouk/Papers/non_op_testing.ps (site visited 2001-05-31)