Software Reliability Growth Models

Alan Wood

Technical Report 96.1
September 1996
Part Number: 130056

Software Reliability Growth Models

Alan Wood
Tandem Computers
10300 N Tantau Ave. LOC55-52
Cupertino, CA 95014
e-mail: wood_alan@tandem.com

Summary

Software reliability is a critical component of computer system availability, so it is important that Tandem's customers experience a small number of software failures in their production environments. Software reliability growth models can be used as an indication of the number of failures that may be encountered after the software has shipped and thus as an indication of whether the software is ready to ship. These models use system test data to predict the number of defects remaining in the software. Software reliability growth models have been applied to portions of several releases at Tandem over the past few years. This experimental research has provided some insights into these models and their utility. The utility of a software reliability growth model is related to its stability and predictive ability. Stability means that the model parameters should not significantly change as new data is added. Predictive ability means that the number of remaining defects predicted by the model should be close to the number found in field use. The major results from the research are:

• While still in the experimental stage, software reliability growth models can be used at Tandem to provide reasonable predictions of the number of defects remaining in the field. The model results are shown below and appear to be extremely good. However, no single methodology has been consistently used to make the predictions. We have had to modify the data or model technique for each different release, e.g., developing a special 2-stage model for release 2 and 3, so no single methodology seems capable of capturing all the variability of different releases.

Release	Predicted Residual Defects	Defects in First Year
1	33	34
2/3	33	28
4	10	9

• Simple models perform as well or better than complex models. We evaluated 9 different software reliability growth models that appear in the literature, and the simple exponential model outperformed the other models in terms of both stability and predictive ability.
• Execution (CPU) time is the best measure of the amount of testing. Using calendar time or number of test cases to measure the amount of testing did not provide credible results.
• Problem reports were a good surrogate for defects. This enhances our ability to make real-time decisions during the test phase because we do not have to wait until problems can be analyzed to determine if they are new defects or rediscoveries of known defects.
• Grouped (weekly) data was sufficient for the models. There is no need to have daily logs of defects and execution time.

Tandem Technical Report 96.1
Part Number 130056
© Tandem Computers, 1996

Table of Contents

Section Number	Section Title	Page Number
1.0	Introduction	1
1.1	Background	2
1.2	Approach and Organization	3
2.0	Software Reliability Growth Models	4
2.1	Software Reliability Growth Model Data	5
2.1.1	Defect Data	5
2.1.2	Test Time Data	5
2.1.3	Grouped Data	6
2.2	Software Reliability Growth Model Types	7
2.3	Parameter Estimation	11
2.3.1	Maximum Likelihood	12
2.3.2	Classical Least Squares	12
2.3.3	Alternative Least Squares	13
2.3.4	Solution Techniques and Hints	13
2.3.5	Theoretical Comparison of Techniques	14
2.4	Definition of a Useful Model	15
3.0	Model Applications	15
3.1	Test Data	16
3.2	Results From the Standard Model	17
3.3	Results for Different Representations of Test Time	20
3.4	Results From Modeling Problem Reports	22
3.5	Results for Different Models	23
3.6	Different Correlation Techniques	24
3.7	Grouped Data Stability	25
3.8	Rerun Test Hours	26
Acknowledgment		27
References		27
Appendix 1	Least Squares Calculations	28
Appendix 2	Parameter Scaling	29

List of Figures

Figure Number	Figure Title	Page Number
1-1.	Residual Defects	2
2-1.	Example Defect Detection Data	4
2-2.	Concave and S-Shaped Models	7
2-3.	Two Stage Model Transformation	10
3-1.	Test Data for All Releases	17
3-2.	Combined Data for Releases 2 and 3	20

List of Tables

Table Number	Table Title	Page Number
1-1.	Model Parameter Options	4
2-1.	Software Reliability Growth Model Examples	8
2-2.	Software Reliability Model Assumptions	9
3-1.	Test Data	16
3-2.	Release 1 Results	18
3-3.	Release 2 Results	18
3-4.	Release 3 Results	18
3-5.	Release 4 Results	19
3-6.	Model Predictions vs. Field Experience	19
3-7.	Release 4 Results for Calendar Time	21
3-8.	Release 3 Results for Number of Test Cases	21
3-9.	Release 2 Results for TPRs	22
3-10.	Release 3 Results for TPRs	22
3-11.	Release 1 Results for Various Models	23
3-12.	Statistical Technique Comparison	24
3-13.	Confidence Interval Comparison for Release 4	25
3-14.	Release 4 Results for Ungrouped Data	26
3-15.	Release 1 Results for Discounting Rerun Test Hours	26

Software Reliability Growth Models

"All Models are Wrong - Some are Useful."
George E. P. Box

1.0 Introduction

For critical business applications, continuous availability is a requirement. and software reliability is an important component of continuous application availability. Tandem customers expect continuous availability, and our process pair technology protects us from most transient software defects. However, rare kinds of single software defects can cause a system failure [Lee,93]. To avoid these failures and to decrease software support costs, Tandem needs to deliver reliable software.

Developing reliable software is one of the most difficult problems facing the software industry. Schedule pressure, resource limitations, and unrealistic requirements can all negatively impact software reliability. Developing reliable software is especially hard when there is interdependence among the software modules as is the case with much of existing software. It is also a hard problem to know whether or not the software being delivered is reliable. After the software is shipped, its reliability is indicated by from customer feedback - problem reports, system outages, complaints or compliments, and so forth. However, by then it is too late; software vendors need to know whether their products are reliable before they are shipped to customers. Software reliability models attempt to provide that information.

There are essentially two types of software reliability models - those that attempt to predict software reliability from design parameters and those that attempt to predict software reliability from test data. The first type of models are usually called "defect density" models and use code characteristics such as lines of code, nesting of loops, external references, input/outputs, and so forth to estimate the number of defects in the software. The second type of models are usually called "software reliability growth" models. These models attempt to statistically correlate defect detection data with known functions such as an exponential function. If the correlation is good, the known function can be used to predict future behavior. Software reliability growth models are the focus of this report.

Most software reliability growth models have a parameter that relates to the total number of defects contained in a set of code. If we know this parameter and the current number of defects discovered, we know how many defects remain in the code (see Figure 1-1). Knowing the number of residual defects helps us decide whether or not the code is ready to ship and how much more testing is required if we decide the code is not ready to ship. It gives us an estimate of the number of failures that our customers will encounter when operating the software. This estimate helps us to plan the appropriate levels of support that will be required for defect correction after the software has shipped and determine the cost of supporting the software.

Software reliability growth models have been applied to portions of four software releases at Tandem over the past 4 years. This research, while still experimental, has provided a number of useful results and insights into software reliability growth modeling. This report describes the methodology we used and the results obtained from modeling Tandem software.

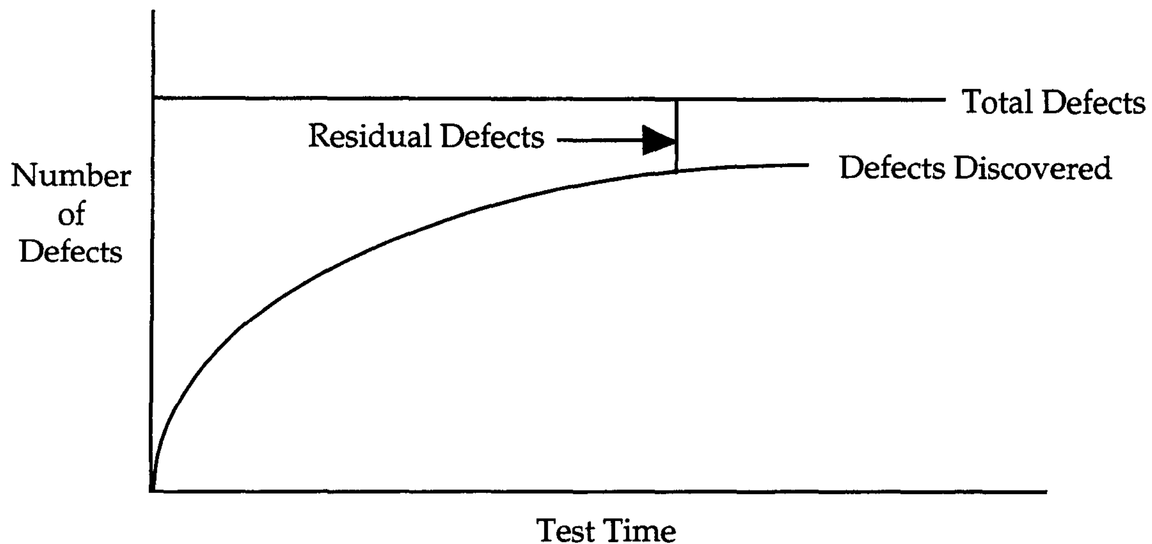


Figure 1-1. Residual Defects

1.1 Background

Software quality is very important at Tandem, but being a commercial company, we don't follow the rigid defense and aerospace software development process, e.g., DOD2167A. We do, however, have a product life cycle, complete with product requirements documents, external and internal specifications, design reviews, code inspections, and so forth. Each product development group has a development organization and a quality assurance (QA) organization. The development organization is responsible for developing the software and performing unit tests. The QA organization is responsible for developing test software and performing integration and system test. When the development organization is satisfied with the functionality and quality of the product, it delivers the software to the QA organization for test. This release milestone is called release to QA (RQA). When the QA organization is satisfied with the functionality and quality of the product, it approves the software for shipment, a milestone called QAOK.

Failures found in the development or test process are reported using Tandem Problem Reports (TPRs). In theory all failures should be reported using TPRs. However, when coding and unit testing the software, it might take a developer longer to submit a TPR than to fix the problem and retest. Therefore, developers do not usually submit TPRs for their own products until after RQA. After a product has reached the RQA point, all failures of that product are required to be reported via TPRs. Therefore, we have good defect data reporting after RQA. During development or test, a product group may experience a failure of another product. These failures are also reported using TPRs, so TPRs may come from sources other than the product test group.

TPRs are reported against the release that is being tested. It is possible that a defect was put into the software in a previous release and was not found until a later release, but it is very difficult to determine the exact time when an error was made. It is also possible that the original code was not defective until other parts of the code changed. Reporting defects against the release in which they are found may over count defects that should have been attributed to a previous release, but it should similarly under count defects from the current release that are not found until later releases. It also replicates the way customers view the release.

There are different types of Tandem software releases. Some are large modifications to many products, some are minor modifications to some products, and some are defect repair for only one or a few products. The major software releases follow the product life cycle process and have large, coordinated QA efforts from most QA groups. We applied software reliability modeling to these major releases. Interim releases and defect repair do not always follow a complete life cycle and QA process. The processes are tailored to fit the amount of change in the release. There is considerable variability in these types of releases, and we did not attempt to model them.

There are many different software reliability growth models, and many different ways to represent the data that is used to create those models. The current software reliability literature is inconclusive as to which models and techniques are best, and some researchers believe that each organization needs to try several approaches to determine what works best for them. This effort is an ongoing experimental effort that attempts to determine the best approach for Tandem. For the models to be useful, they must add value to the corporate decision making process, e.g., they are used to help determine when to ship a product or size the sustaining effort after the product is shipped. In order for the results to be used for these types of corporate decisions, the results must be stable and must predict the field failure rate with reasonable accuracy (see Section 2.5).

Terminology

As defined by IEEE, an *error* is a human action that results in a *fault*. Encountering a fault during system operation can cause a *failure*. A *bug* is synonymous with fault, and a *defect* is very similar. This paper follows those definitions. The words defect and bug are used to mean code that does not satisfy the user requirements, either because a requirement is incorrectly designed or implemented (the vast majority) or was not implemented. A failure is what a customer or tester encountered that caused them to report the defect.

1.2 Approach and Organization

Software reliability growth models have been applied to portions of several releases over the past few years. Since this document is accessible by non-Tandem employees, the specific products and releases are not mentioned and the data has been appropriately transformed. The models have been applied to the QA test period - from RQA to QAOK. The purpose of this report is to describe Tandem's experience with reliability growth models over the past few years. The report contains the data we gathered and the techniques used to analyze the data. Section 2 presents software reliability growth model theory while Section 3 describes Tandem's experience with various models and techniques. Section 2.1 describes the data necessary for software reliability growth models, and Section 2.2 presents several model types. In Section 2.3, different statistical techniques for model parameter evaluation are presented. Section 2.4 introduces some ideas for evaluating model utility. Section 3.1 presents the raw data used for the models. Section 3.2 presents the basic results we achieved and our evaluation of the model utility. Sections 3.3-3.8 present the results obtained by varying model parameters.

There are many variables to be considered in developing a software reliability growth model methodology. These variables are listed in Table 1-1. We have experimented with many combinations of these variables over the last few years, and the results of those experiments are also shown in Table 1-1. The sections of the report that describe the parameters and contain the results of varying them are also shown in the table.

Parameter	Options	Result Summary	Report Section
Amount of Testing	Execution (CPU) time Calendar time Number of test cases	Execution time is the only option that works	2.1.1, 3.3
Defect Data	Defects TPRs	TPRs are a good surrogate for defects	2.1.2, 3.4
Grouped Data	Defect occurrence time Weekly (grouped) summary	Grouped data works fine and is easier to collect	2.1.3, 3.7
Growth Model Type	Many (Table 2-1)	Simple exponential (G-O) model works best	2.2, 3.5
Statistical Technique	Maximum likelihood Classical least squares Alternative least squares	Alternative least squares provides the best point estimates. Maximum likelihood is preferred for confidence intervals.	2.3, 3.6

Table 1-1. Model Parameter Options

2.0 Software Reliability Growth Models

Reliability is usually defined as the probability that a system will operate without failure for a specified time period under specified operating conditions. Reliability is concerned with the time between failures or its reciprocal, the failure rate. In this report we are considering data from a test environment, so we report defect detection rate rather than failure rate. A defect detection is usually a failure during a test, but test software may also detect a defect even though the test continues to operate. Defects can also be detected during design reviews or code inspections, but we do not consider those sorts of activities in this report. Time in a test environment is a synonym for amount of testing, which can be measured in several ways. Defect detection data consists of a time for each defect or group of defects and can be plotted as shown in Figure 2-1. We can derive defect detection rates from this data.

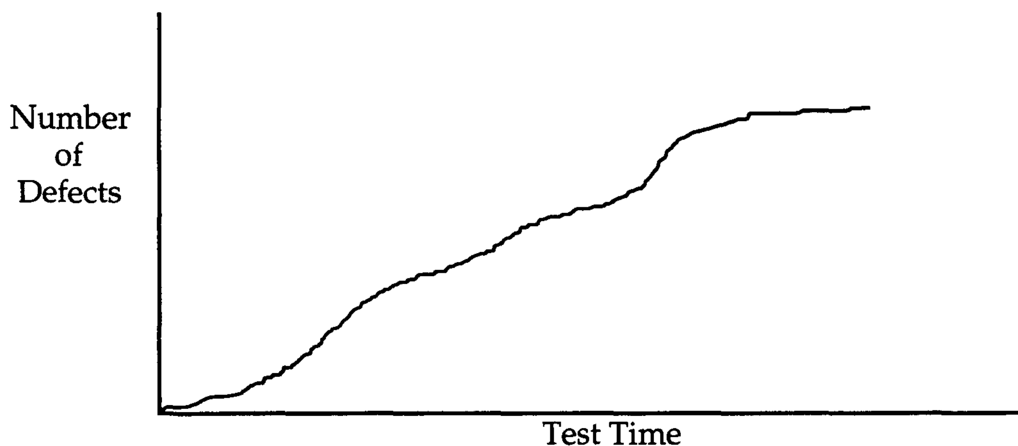


Figure 2-1. Example Defect Detection Data

A cumulative plot of defects vs amount of testing such as Figure 2-1 should show that the defect discovery rate decreases as the amount of testing increases. The theory is that each defect is fixed as it is discovered. This decreases the number of defects in the code, so the defect discovery rate should decrease (the length of time between defect discoveries should increase). When the defect discovery rate reaches an acceptably low value, the software is deemed suitable to ship. However, it is difficult to extrapolate from defect discovery rate in a test environment to failure rate during system operation, primarily because it is hard to extrapolate from test time to system operation time. Instead, we look at the expected quantity of remaining defects in the code. These residual defects provide an upper limit on the number of unique failures our customers could encounter in field use.

Software reliability growth models are a statistical interpolation of defect detection data by mathematical functions. The functions are used to predict future failure rates or the number of residual defects in the code. There are different ways to represent defect detection data as discussed in Section 2.1. There are many types of software reliability growth models as described in Section 2.2, and there are different ways to statistically correlate the data to the models as discussed in Section 2.3. Current software reliability literature is inconclusive as to which data representation, software reliability growth model, and statistical correlation technique works best. The advice in the literature seems to be to try a number of the different techniques and see which works best in your environment. In Section 3, we describe the application of the techniques in the Tandem environment.

2.1 Software Reliability Growth Model Data

There are two relevant types of data for software reliability growth models. The first is the time at which the defect was discovered, discussed in Section 2.1.1, and the second is the number of defects discovered, discussed in Section 2.1.2. Aggregated or grouped data is described in Section 2.1.3.

2.1.1 Test Time Data

For a software reliability growth model developed during QA test, the appropriate measure of time must relate to the testing effort. There are three possible candidates for measuring test time:

- calendar time
- number of tests run
- execution (CPU) time.

Test time can simply be calendar time, particularly if there is a dedicated test group that continuously runs the test machines. However, the test effort is often asynchronous, so number of tests run or execution time is normally used in place of calendar time. Number of tests run would be a good measure if all tests had a similar probability of detecting a defect, but often that is not the case. We have some test suites that execute 100 tests in an hour and other more sophisticated tests that take 24 hours to execute. The longer test cases usually stress the software more and thus have a higher probability of finding a defect per test case run. We have developed software reliability growth models using calendar time, number of tests, and execution (CPU) time as a measure of time. The results, showing that execution time is the best measure of test time, are described in Section 3.3.

2.1.2 Defect Data

At Tandem, potential defect discoveries are recorded as Tandem Problem Reports (TPRs). TPRs are analyzed by software developers to determine if a new defect has been detected. TPRs are not always defects because they may represent a non-defect and because multiple people or groups may discover the same defect. Non-defect TPRs can represent confusion

about how to use a set of software or what the software is supposed to produce. These TPRs are not counted as defects and are usually closed with a notation that a question has been answered or that the software performed as expected. Defects that have been found by multiple people or groups are usually called duplicates or rediscoveries. Rediscoveries are not included in the defect counts since the original defect report is counted. TPRs that do not represent new defects (non-defects and rediscoveries) are called "smoke" by software developers. The amount of smoke generated during QA test varies over time and by release, but 30-40% of all TPRs is a good estimate of the number of smoke TPRs. The large percentage of smoke TPRs is caused by significant parallel usage for the products under test, resulting in duplicate TPRs. The amount of smoke after the software is shipped to customers is usually higher since many different customers may encounter the same failure.

TPRs are classified as to the severity of the defect. The severities range from 3 (most severe) to 0 (least severe). The severity levels are assigned depending on how urgently the customer needs a solution as shown below.

Severity	Customer Impact	
0	No Impact:	Can tolerate the situation indefinitely.
1	Minor:	Can tolerate the situation, but expect solution eventually.
2	Major:	Can tolerate the situation, but not for long. Solution needed.
3	Critical:	Intolerable situation. Solution urgently needed.

Only severity 2 and 3 defect data are used for the software reliability growth models. This is because the models are based on QA testing, and test personnel usually only submit severity 2 and 3 TPRs because severity 0 and 1 TPRs do not usually impact their testing. Therefore, it would not be possible to predict severity 0 and 1 defect rates based on the test data.

Defect only TPRs (no smoke) represent the number of unique defects in the code and are thus the appropriate data to use in software reliability growth models. However, it is useful to model total TPRs as a surrogate for defects because it takes time to analyze a TPR and determine if the TPR is a new defect or smoke. Our estimates are that 50% of the TPRs are analyzed within 1 week and 90% are analyzed within 2 weeks. Therefore, reliable defect data lags total TPR data by about 2 weeks. If we are trying to make a decision about shipping the software, we want to use the most current data, not data that is 2 weeks old, so a model based on TPRs is valuable if it provides a reasonable prediction for residual defects. Most of this report describes models that were developed using software defect data. However, in Section 3.4, we describe the results of modeling TPRs instead of unique defects.

2.1.3 Grouped Data

The best possible data would be a list of the failure occurrence times, where time may mean calendar time, execution time, or test case number. Unfortunately, we are only able to gather weekly or "grouped" data, that is, we know the amount of failures and test time that occurred during a week. TPRs have a time stamp that indicates when they are filed, but QA personnel sometimes batch their work and may not submit the TPRs found during a week of testing until the end of the week. Therefore, the TPR time stamps are unreliable on a daily basis but are reliable on a weekly basis. We have done experiments by randomizing our weekly data to create exact failure occurrence times, and it appears that the model results are the same for either grouped data or exact failure occurrence times (see Section 3.7).

2.2 Software Reliability Growth Model Types

Software reliability growth models have been grouped into two classes of models - concave[1] and S-shaped. These two model types are shown in Figure 2-2. The most important thing about both models is that they have the same asymptotic behavior, i.e., the defect detection rate decreases as the number of defects detected (and repaired) increases, and the total number of defects detected asymptotically approaches a finite value. The theory for this asymptotic behavior is that:

(1) A finite amount of code should have a finite number of defects. Repair and new functionality may introduce new defects, which increases the original finite number of defects. Some models explicitly account for new defect introduction during test while others assume they are negligible or handled by the statistical fit of the software reliability growth model to the data.

(2) It is assumed that the defect detection rate is proportional to the number of defects in the code. Each time a defect is repaired, there are fewer total defects in the code, so the defect detection rate decreases as the number of defects detected (and repaired) increases. The concave model strictly follows this pattern. In the S-shaped model, it is assumed that the early testing is not as efficient as later testing, so there is a ramp-up period during which the defect detection rate increases. This could be a good assumption if the first QA tests are simply repeating tests that developers have already run or if early QA tests uncover defects in other products that prevent QA from finding defects in the product being tested. For example, an application test may uncover OS defects that need to be corrected before the application can be run. Application test hours are accumulated, but defect data is minimal because OS defects don't count as part of the application test data. After the OS defects are corrected, the remainder of the application test data (after the inflection point in the S-shaped curve) looks like the concave model.

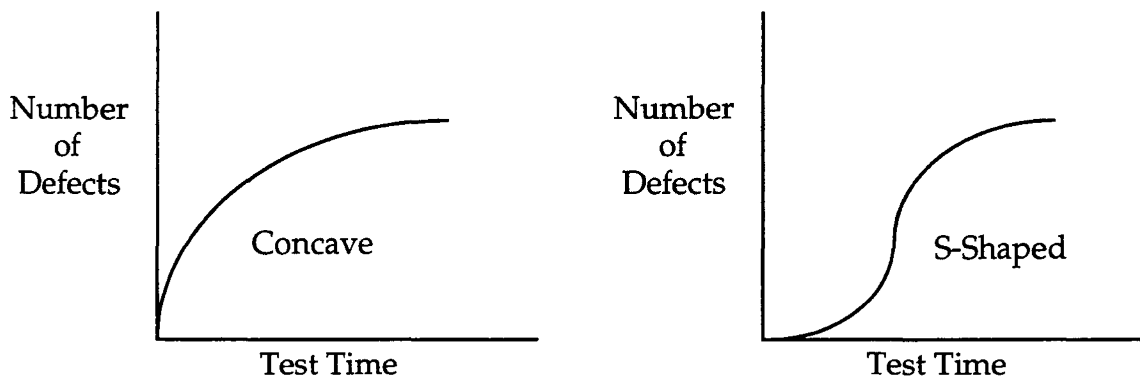


Figure 2-2. Concave and S-Shaped Models

There are many different representations of software reliability models. In this paper we use the model representation shown in Figure 2-2. This representation shows the expected number of defects at time t and is denoted $\mu(t)$, where t can be calendar time, execution time, or number of tests executed as described in Section 2.1. An example equation for $\mu(t)$ is the Goel-Okumoto (G-O) model:

[1] The word concave is used for this class of models because they are all concave functions, i.e., continually bending downward. Functions that bend upward are called convex functions. S-shaped functions are first convex and then concave.

$\mu(t) = a(1-e^{-bt})$, where
a = expected total number of defects in the code
b = shape factor = the rate at which the failure rate decreases, i.e., the rate at which we approach the total number of defects.

The Goel-Okumoto model is a concave model, and the parameter "a" would be plotted as the total number of defects in Figure 2-2. The Goel-Okumoto model has 2 parameters; other models can have 3 or more parameters. For most models, $\mu(t) = aF(t)$, where a is the expected total number of defects in the code and F(t) is a cumulative distribution function. Note that $F(0) = 0$, so no defects are discovered before the test starts, and $F(\infty) = 1$, so $\mu(\infty) = a$ and a is the total number of defects discovered after an infinite amount of testing. Table 2-1 provides a list of the models that were evaluated as part of this effort. A derivation of the properties of most of these models can be found in [Musa,87].

Model Name	Model Type	$\mu(t)$	Reference	Comments
Goel-Okumoto (G-O)	Concave	$a(1-e^{-bt})$ $a\geq 0, b>0$	Goel,79	Also called Musa model or exponential model
G-O S-Shaped	S-Shaped	$a(1-(1+bt)e^{-bt})$ $a\geq 0, b>0$	Yamada,83	Modification of G-O model to make it S-shaped (Gamma function instead of exponential)
Hossain-Dahiya/G-O	Concave	$a(1-e^{-bt})/(1+ce^{-bt})$ $a\geq 0, b>0, c>0$	Hossain,93	Solves a technical condition with the G-O model. Becomes same as G-O as c approaches 0.
Gompertz	S-Shaped	$a(b^{c^t})$ $a\geq 0, 0\leq b\leq 1, 0<c<1$	Kececioglu, 91	Used by Fujitsu, Numazu Works
Pareto	Concave	$a(1-(1+t/\beta)^{1-\alpha}$ $a\geq 0, \beta>0, 0\leq\alpha\leq 1$	Littlewood, 81	Assumes failures have different failure rates and failures with highest rates removed first
Weibull	Concave	$a(1-e^{-bt^c})$ $a\geq 0, b>0, c>0$	Musa,87	Same as G-O for c=1
Yamada Exponential	Concave	$a(1-e^{-r\alpha(1-e^{-\beta t})})$ $a\geq 0, r\alpha>0, \beta>0$	Yamada,86	Attempts to account for testing effort
Yamada Raleigh	S-Shaped	$a(1-e^{-r\alpha(1-e^{(-\beta t^2/2)})})$ $a\geq 0, r\alpha>0, \beta>0$	Yamada,86	Attempts to account for testing effort
Log Poisson	Infinite Failure	$(1/c)\ln(c\alpha t+1)$ $c>0, \alpha>0$	Musa,87	Failure rate decreases but does not approach 0

Table 2-1. Software Reliability Growth Model Examples

The Log Poisson model is a different type of model. This model assumes that the code has an infinite number of failures. Although this is not theoretically true, it may be essentially true in practice since all the defects are never found before the code is rewritten, and the model may provide a good fit for the useful life of the product.

The models all make assumptions about testing and defect repair. Some of these assumptions seem very reasonable, but some are questionable. Table 2-2 contains a list and discussion of these assumptions.

Assumption	Reality
Defects are repaired immediately when they are discovered	Defects are not repaired immediately, but this can be partially accommodated by not counting duplicates. Test time may be artificially accumulated if a non-repaired defect prevents other defects from being found.
Defect repair is perfect	Defect repair introduces new defects. The new defects are less likely to be discovered by test since the retest for the repaired code is not usually as comprehensive as the original testing.
No new code is introduced during QA test	New code is frequently introduced throughout the entire test period, both defect repair and new features. This is accounted for in parameter estimation since actual defect discoveries are used, but may change the shape of the curve, i.e., make it less concave. The multi-stage model, discussed in Section 2.4, is an attempt to account for new code introduction.
Defects are only reported by the product testing group	Defects are reported by lots of groups because of parallel testing activity. If we add the test time for those groups, we have the problem of equivalency between an hour of QA test time and an hour of test time from a group that is testing a different product. This can be accommodated by restricting defects to those discovered by QA, but that eliminates important data. This problem means that defects do not correlate perfectly with test time.
Each unit of time (calendar, execution, number of test cases) is equivalent	This is certainly not true for calendar time or test cases as discussed earlier. For execution time, "corner" tests sometimes are more likely to find defects, so those tests create more stress on a per hour basis. When there is a section of code that has not been as thoroughly tested as other code, e.g., a product that is under schedule pressure, tests of that code will usually find more defects. Many tests are rerun to ensure defect repair has been done properly, and these reruns should be less likely to find new defects. However, as long as test sequences are reasonably consistent from release to release, this can be accounted for if necessary from lessons learned on previous releases.
Tests represent operational profile	Customers run so many different configurations and applications that it is difficult to define an appropriate operational profile. In some cases, the sheer size and transaction volume of the production system makes the operational environment impractical to replicate. The tests contained in the QA test library test basic functionality and operation, error recovery, and specific areas with which we have had problems in the past. Additional tests are continually being added, but the code also learns the old tests, i.e., the defects that the old tests would have uncovered have been repaired.
Failures are independent	Our experience is that this is reasonable except when there is a section of code that has not been as thoroughly tested as other code, e.g., a product behind schedule that was not thoroughly unit tested. Tests run against this section of code may find a disproportionate share of defects. [Musa,87,P242] has a detailed discussion of the independence assumption.

Table 2-2. Software Reliability Model Assumptions

It is difficult to determine how the violation of the model assumptions will affect the models. For instance, introducing new functionality may make the curve less concave, but test reruns could make it more concave. Removing defects discovered by other groups comes closer to satisfying the model assumptions but makes the model less useful because we are not including all the data (which may also make the results less statistically valid). In general, small violations probably get lost in the noise while significant violations may force us to revise the models, e.g., see the discussion of Release 4 test hours at the end of Section 3.1. Given the uncertainties about the effects of violating model assumption, the best strategy is to try the models to see what works best for a particular style of software development and test.

Multi-Stage Models

One of the assumptions made by all the models is that the set of code being testing is unchanged throughout the test period. Clearly, defect repair invalidates that assumption, but it is assumed that the effects of defect repair are minimal so that the model is still a good approximation. If a significant amount of new code is added during the test period, there is a technique that allows us to translate the data to account for the increased code change. Theoretically, the problem is that adding a significant amount of changed code should increase the defect detection rate. Therefore, the overall curve will look something like Figure 2-3, where D_1 defects are found in T_1 time prior to the addition of the new code and an additional D_2-D_1 defects are found in T_2-T_1 time after that code addition. The problem is to translate the data to a model $\mu(t)$ that would have been obtained if the new code had been part of the software at the beginning of the test. This translation is discussed in Chapter 15 of [Musa,87]. Let $\mu_1(t)$ model the defect data prior to the addition of the new code, and let $\mu_2(t)$ model the defect data after that code addition. The model $\mu(t)$ is created by appropriately modifying the failure times from $\mu_1(t)$ and $\mu_2(t)$. This section describes how to perform the translation assuming $\mu(t)$, $\mu_1(t)$, and $\mu_2(t)$ are all G-O models. In theory, this technique could be applied to any of the models in Table 2-1, including the S-shaped models.

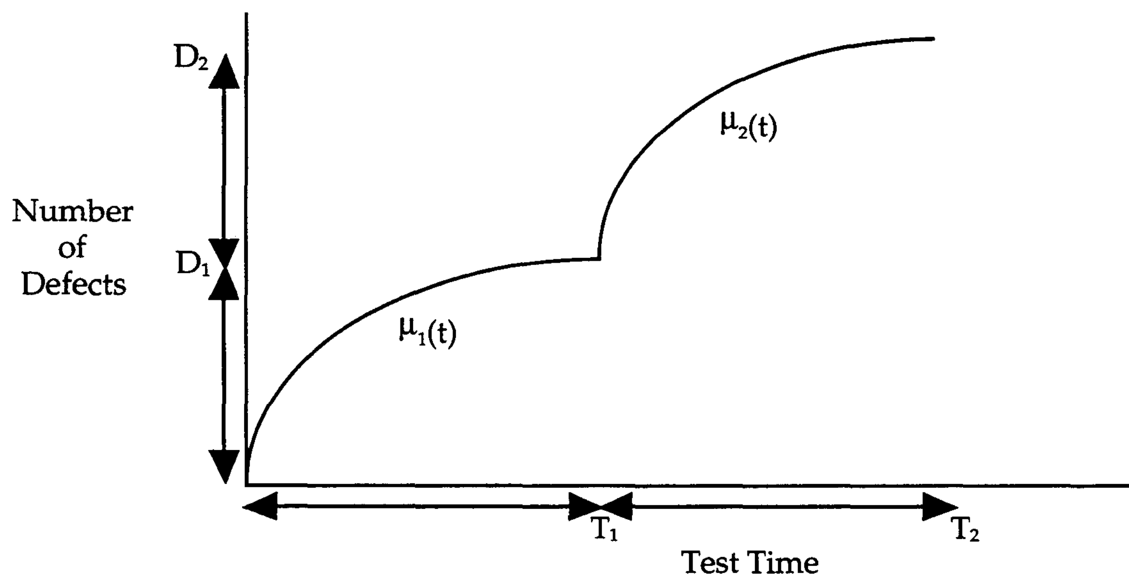


Figure 2-3. Two Stage Model Transformation

Assume that model $\mu_1(t)$ applies to the time period 0-T_1 and that model $\mu_2(t)$ applies to the time period from T_1-T_2 as shown in Figure 2-3. The first step in the translation is to determine the parameters of the models $\mu_1(t)$ and $\mu_2(t)$ to get $\mu_1(t) = a_1(1-e^{-b_1t})$ and $\mu_2(t) = a_2(1-e^{-b_2t})$. The calculations for $\mu_1(t)$ are the standard techniques described in Section 2.3 using the data in time period 0-T_1. The calculations for $\mu_2(t)$ are also the standard techniques assuming that the test started at time T_1 and produced D_2-D_1 defects. In other words, subtract D_1 from the cumulative defects and subtract T_1 from the cumulative time when calculating $\mu_2(t)$.

The next step is to calculate the translated time for the defects observed prior to the insertion of the new code. The time for each defect is translated according to the following equation (equation 15.18 in [Musa,87]).

$$\tau_i = (-1/b_2)\ln\{1-(a_1/a_2)(1-e^{-b_1t_i})\}$$

Next, calculate the translated time for the defects observed after the insertion of the new code. Start by calculating the expected amount of time it would have taken to observe D_1 defects if the new code had been part of the original code released at the start of the test.[2] This time τ is calculated from $D_1 = a_2(1-e^{-b_2\tau})$ or $\tau = (-1/b_2)\ln\{1-D_1/a_2\}$. This time should be shorter than T_1 because the failure rate would have been higher at the start of test if there were more defects at the start of test. Then all failure times from the T_1-T_2 time period are translated by subtracting T_1-τ, i.e., $\tau_i = t_i-(T_1-\tau)$. This essentially translates the defect times in the T_1-T_2 time period to the left, meaning that we would have expected to have found more defects earlier if there were more to find at the beginning of the test.

Finally, we use the standard techniques from Section 2.3 to determine the parameters *a* and *b* in $\mu(t) = a(1-e^{-bt})$, where the defect times are adjusted as described previously. The adjustments made to the failure times provide the failure times that would have theoretically been observed if the new code had been released at the beginning of the test rather than part of the way through the test.

2.3 Parameter Estimation

A software reliability model is a function such as those shown in Table 2-1. Fitting this function to the data means estimating its parameters from the data. One approach to estimating parameters is to input the data directly into equations for the parameters. The most common method for this direct parameter estimation is the maximum likelihood technique described in Section 2.3.1. A second approach is fitting the curve described by the function to the data and estimating the parameters from the best fit to the curve. The most common method for this indirect parameter estimation is the least squares technique. The classical least squares technique is described in Section 2.3.2. and an alternative squares technique is described in Section 2.3.3. The alternative least squares technique was used most often since it provided the best results. A comparison of the results obtained by using each of these techniques is described in Section 3.6.

---

[2] Actually, this step is slightly more complicated. D_1 is replaced by the expected number of defects observed in T_1 according to model $\mu_1(t)$, i.e., D_1 is replaced by $\mu_1(T_1)$. In our experience, D_1 and $\mu_1(T_1)$ are essentially identical.

2.3.1 Maximum Likelihood

The maximum likelihood technique consists of solving a set of simultaneous equations for parameter values. The equations define parameter values that maximize the likelihood that the observed data came from a distribution with those parameter values. Maximum likelihood estimation satisfies a number of important statistical conditions for an optimal estimator and is generally considered to be the best statistical estimator for large sample sizes. Unfortunately, the set of simultaneous equations it defines are very complex and usually have to be solved numerically. For a general discussion of maximum likelihood theory and equation derivation, see [Mood,74] and [Musa,87]. Here, we only show the equations that must be solved to provide parameter estimates and confidence intervals for the Goel-Okumoto (G-O) model.

The expected number of defects for the G-O model is

$\mu(t) = a(1-e^{-bt})$, where
a = expected total number of defects in the code
b = shape factor = the rate at which the failure rate decreases.

From Equation (12.117) of [Musa,87], the parameter *b* can be estimated by solving:

(1) $\Sigma_{i=1}^{w}(f_i - f_{i-1})(t_i e^{-bt_i} - t_{i-1}e^{-bt_{i-1}})/(e^{-bt_i} - e^{-bt_{i-1}}) = f_w t_w/(1 - e^{bt_w})$, where
w = current number of weeks of QA test
t_i = cumulative test time at the end of the ith week
f_i = cumulative number of failures at the end of the ith week.

From Equation (12.134) of [Musa,87], the α per cent confidence interval (e.g., 95%) for *b* is given by:

(2) $b \pm Z_{1-\alpha/2}/(I_0(b))^{0.5}$, where
$Z_{1-\alpha/2}$ is the value of the standard Normal, e.g., 1.645 for 90% confidence interval,
$I_0(b) = \Sigma_{i=1}^{w}(f_i - f_{i-1})(t_i - t_{i-1})^2 e^{-b(t_i + t_{i-1})}/(e^{-bt_{i-1}} - e^{-bt_i})^2 - f_w t_w^2 e^{bt_w}/(e^{bt_w} - 1)^2$

The parameter *a* and its confidence interval can then be estimated by solving:

(3) $a = f_w/(1 - e^{-bt_w})$, where *b* is one of the values obtained above.

When implementing these equations, equation (1) is solved numerically to derive an estimate of *b*, and the appropriate confidence interval for *b* is then calculated from (2). The parameter *a* is then estimated using (3) and the estimate of *b*. Confidence intervals for *a* are calculated using (3) and the values from (2).

2.3.2 Classical Least Squares

The maximum likelihood technique solves directly for the optimal parameter values. The least squares method solves for parameter values by picking the values that best fit a curve to the data. This technique is generally considered to be the best for small to medium sample sizes. The theory of least squares curve-fitting (see [Mood,74]) is that we want to find parameter values that minimize the "difference" between the data and the function fitting the data, where the difference is defined as the sum of the squared errors. The classical least squares technique involves log likelihood functions and is described in [Musa,87,Section 12.3]. From [Musa,87,Equation 12.141], the expression to be minimized for the G-O model is

(4) $\Sigma_{i=1}^{w}(\ln((f_i-f_{i-1})/(t_i-t_{i-1}))- \ln(b) - \ln(a-f_i))^2$, where, as in Section 2.3.1,
w = current number of weeks of QA test
t_i = cumulative test time at the end of the ith week
f_i = cumulative number of failures at the end of the ith week.

Confidence intervals are given by [Musa, p. 358] as:

(5) $a \pm t_{w-2,\alpha/2}\,(\mathrm{Var}(a))^{0.5}$ where
$t_{w-2,\alpha/2}$ is the upper $\alpha/2$ percentage point of the t distribution with w-2 degrees of freedom
Var(a) is the variance of *a*; calculation of the variance is described in Appendix 1.

The confidence interval for *b* is the same as the above equation with *b* replacing *a*. These confidence intervals are derived by assuming that *a* and *b* are normally distributed. Note that the confidence intervals are symmetric in contrast to the asymmetric confidence intervals provided by maximum likelihood.

2.3.3 Alternative Least Squares

An alternative approach to least squares is to directly minimize the difference between the observed number of failures and the predicted number of failures. For this approach the quantity to be minimized is:

(6) $\Sigma_{i=1}^{w}(f_i - \mu(t_i))^2$, where, as in Section 2.3.1,
w = current number of weeks of QA test
$\mu(t_i)$ = the cumulative expected number of defects at time t_i
t_i = cumulative test time at the end of the ith week
f_i = cumulative number of failures at the end of the ith week.

This technique is easy to use for any software reliability growth model since the minimization can be done by an optimization package such as the Solver in Microsoft® Excel. It is not normally described in textbooks because it does not lead to a set of equations that can be solved, but with the increased availability of optimization packages, the minimization can be solved directly instead of reducing it to a set of equations. Note that any of software reliability growth models from Table 2-1 can be used in this equation by using the appropriate $\mu(t)$. For the G-O model, Equation (6) becomes:

(7) $\Sigma_{i=1}^{w}(f_i - a(1-e^{-bt_i}))^2$.

Confidence intervals for the parameters in Equation (7) are the same as for classical least squares and are given by Equation (5). The calculations required for these confidence intervals is described in Appendix 1. Note that these confidence intervals are symmetric.

2.3.4 Solution Techniques and Hints

The Solver in Microsoft® Excel was used to solve the minimizations defined in the preceding sections. However, since these are non-linear equations, the solution found may not be appropriate (a local optimum rather than a global optimum) or it may not be possible to determine a solution in a reasonable amount of time. To help avoid this problem, it is

useful to define parameter values that are close to the final values. This may require some experimentation prior to running the optimization. If a solution has been obtained using the previous week's data, those parameter values are usually a good starting point. If this is the first attempt to solve the minimization, parameter values should be selected that provide a reasonable match to the existing data. This is easy to do in a spreadsheet with one column of data and a second column of predicted values based on a given function and the chosen parameter values.

Transforming the test hour data should not affect the total number of defects parameter. However, before the parameters become stable, transforming the test hour data may help numerical stability. For example, we used per cent of planned test hours completed rather than actual test hours completed in week 7 for Release 3 (one week before we began to get parameter stability). Per cent test hours predicted 400 total defects while actual test hour data predicted 5000 total defects. Neither answer is close to the right value of about 100, but using per cent test hours was closer, and the solution was reached much more quickly.

2.3.5 Theoretical Comparison of Techniques

This section compares the three parameter estimation techniques from a theoretical perspective. We focus on their ease of use, confidence interval shape, and parameter scalability. The more important comparison of model stability and predictive ability on actual data is contained in Section 3.6.

Since optimization packages are readily available, Equations (1) - maximum likelihood, (4) - classical least squares, and (6) - alternative least squares are all straightforward to solve. However, Equation (1) only applies to the G-O model, and a new maximum likelihood equation must be derived for each software reliability growth model. These equations can be difficult to derive, especially for the more complex models. Equation (4) applies to the exponential family of models that includes the G-O model. It is fairly easy to modify this equation for similar models. Equation (6) is the easiest to use since it applies to any software reliability growth model, so the alternative least squares method is the easiest to apply.

Confidence intervals for all of the estimation techniques are based on assuming that estimation errors are normally distributed. For the maximum likelihood technique, this assumption is good for large sample sizes because of the asymptotically normal properties of this estimator. However, it is not as good for the smaller samples that we typically have. Nevertheless, the maximum likelihood technique provides the best confidence intervals because it requires less normality assumptions and because it provides asymmetric confidence intervals for the total defect parameter. The lower confidence limit is larger than the number of experienced defects, and the upper confidence limit is farther from the point estimate than the lower confidence limit to represent the possibility that there could be many defects that have gone undetected by testing. Conversely, for the least squares techniques, the lower confidence limit can be less than the number of experienced defects (which is obviously impossible), and the confidence interval is symmetric. Also, additional assumptions pertaining to the normality of the parameters is necessary to derive confidence intervals for the least squares techniques.

The transformation technique consists of multiplying the test time by an arbitrary (but convenient) constant and multiplying the number of defects observed by a different arbitrary constant. For this technique to work, the predicted number of total defects must be unaffected by the test time scaling and must scale the by same amount as the defect data. For example, we may experience 50 total defects during test and want to scale that to 100 for confidentiality or ease of reporting. To do that transformation, the number of defects

reported each week must be multiplied by 2. If 75 total defects were predicted by a model based on the unscaled data, then the total defects predicted from the scaled data should be 150. Fortunately, all three of the parameter estimation techniques provide this linear scaling property as shown in Appendix 2. In addition the least squares confidence intervals scale linearly as shown in Appendix 2, but the maximum likelihood confidence intervals do not.

2.4 Definition of a Useful Model

Since none of the models will match any company's software development and test process exactly, what makes the model useful? The answer to this question relates to what we want the model to do. During the test, we would like the model to predict the additional test effort required to achieve a quality level (as measured by number of remaining defects) that we deem suitable for customer use. At the end of the test, we would like the model to predict the number of remaining defects that will be experienced (as failures) in field usage. This leads to two criteria for a useful model: (1) The model must become stable during the test period, i.e., the predicted number of total defects should not vary significantly from week to week, and (2) The model must provide a reasonably accurate prediction of the number of defects that will be discovered in field use.

(1) The model must become stable during the test period and remain stable until the end of the test (assuming the test process remains stable).

If the model says that there are 50 remaining defects one week and 200 the next, no one is going to believe either prediction. For a model to be accepted by management, the predicted number of total defects should not vary significantly from week to week. Stability is subjective, but in our experience a good rule of thumb is that weekly predictions from the model should vary by no more than 10%. Also, the confidence intervals around the total defect parameter should be shrinking. It would be nice if the model was immediately stable, but parameter estimation requires a reasonable amount of data. In particular, the data must begin to show concave behavior since the speed at which the failure rate decreases is critical to estimating the total number of defects in the code. The literature (e.g., [Musa,87,P.194,P.311] and [Ehrlich,90,P.63]) and our experience indicate that the model parameters do not become stable until about 60% of the way through the test. This is sufficient since management will not be closely monitoring the model until near the end of expected test completion.

(2) The model must provide a reasonably accurate prediction of the number of defects that will be discovered in field use.

Since field use is very different from a test environment, no model derived from the test environment can expect to be perfectly accurate. However, if the number of defects is within the 90% confidence levels developed from the model, the model is reasonably accurate. Unfortunately, the range defined by the 90% confidence levels may be much larger than software development managers would like. In our experience, 90% confidence intervals are often larger than twice the predicted residual defects.

3.0 Model Applications

Over the past few years, we have collected test data from a subset of products for four software releases. To avoid confidentiality issues, the specific products and releases are not identified, and the test data has been suitably transformed. The literature has very little real data from commercial applications, possibly due to confidentiality concerns. We hope this transformation technique will stimulate other software reliability practitioners to provide similarly transformed data that can be used for model development and testing by theoreticians.

The test data collected included three representations of the amount of testing and two representations of defects as described in Section 2.1. For each of the software releases, we evaluated the test data using the software reliability growth models described in Section 2.2, the statistical techniques described in Section 2.3, and the model evaluation criteria described in Section 2.4. This section describes the results of those evaluations. Section 3.1 contains the test data, Section 3.2 contains the basic results, and Sections 3.3-3.8 contain results obtained by varying a model parameter or evaluation technique.

3.1 Test Data

We collected data from four separate software releases. As shown in Table 3-1, we artificially set the system test time for Release 1 to 10,000 hours and the number of defects discovered in Release 1 to 100. All data was ratioed proportionately, e.g., all test hours were multiplied by 10,000 and divided by the real number of test hours from Release 1. As mentioned in Section 2.3.5, the predicted number of total defects scales by same amount as the defect data and is unaffected by the test time scaling. The releases were tested for different lengths of time (both calendar and execution) as shown in the table. The data in Table 3-1 is shown graphically in Figure 3-1. All the data exhibits the shape of the concave models, e.g., Figure 1-1.

	Release 1		Release 2		Release 3		Release 4	
Test Week	Execu-tion hrs	No. of defects	Execu-tion hrs	No. of defects	Execu-tion hrs	No. of defects	Execu-tion hrs	No. of defects
1	519	16	384	13	162	6	254	1
2	968	24	1,186	18	499	9	788	3
3	1,430	27	1,471	26	715	13	1,054	8
4	1,893	33	2,236	34	1,137	20	1,393	9
5	2,490	41	2,772	40	1,799	28	2,216	11
6	3,058	49	2,967	48	2,438	40	2,880	16
7	3,625	54	3,812	61	2,818	48	3,593	19
8	4,422	58	4,880	75	3,574	54	4,281	25
9	5,218	69	6,104	84	4,234	57	5,180	27
10	5,823	75	6,634	89	4,680	59	6,003	29
11	6,539	81	7,229	95	4,955	60	7,621	32
12	7,083	86	8,072	100	5,053	61	8,783	32
13	7,487	90	8,484	104			9,604	36
14	7,846	93	8,847	110			10,064	38
15	8,205	96	9,253	112			10,560	39
16	8,564	98	9,712	114			11,008	39
17	8,923	99	10,083	117			11,237	41
18	9,282	100	10,174	118			11,243	42
19	9,641	100	10,272	120			11,305	42
20	10,000	100						

Note: all data has been scaled by artificially setting the execution time in Release 1 to 10,000 hours and the number of defects discovered in Release 1 to 100 and ratioing all other data proportionately.

Table 3-1. Test Data

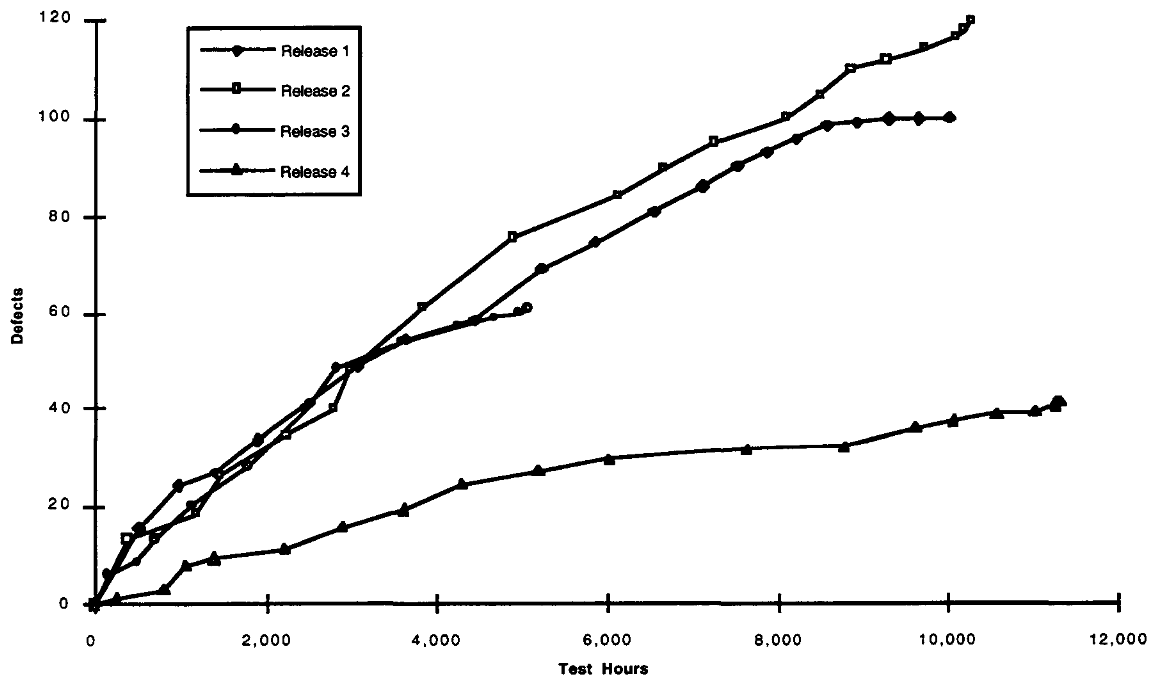


Figure 3-1. Test Data for All Releases

The execution hours for Releases 1-3 are obtained from the product QA groups testing the release subsets used in this report. Other QA and development groups reported defects against the release subsets, but the test effort was reasonably synchronized across all groups, so we feel that the product QA test hours fairly represents the software test effort. For Release 4, the test effort was not well synchronized, and a larger portion of the defects were reported by groups other than the product QA groups. Therefore, we added the test hours from product QA groups that were not directly testing the release subset but were reporting many defects. We feel this is a better representation of the software test effort.

3.2 Results From the Standard Model

As will be shown in the following sections. we achieved the best results using execution time to measure the amount of testing, defect data rather than problem reports, and the G-O (exponential) software reliability growth model. We gathered data weekly and used the alternative least squares described in Section 2.3.3 technique to estimate the parameters. The most important parameter is the predicted total number of defects from which we can determine the predicted number of residual defects. A few of these calculations are shown in Tables 3-2 through 3-5. As can be seen from the tables, the total defect parameter becomes stable (meaning the week to week variance is small) after approximately 60% of calendar test time and 70% of execution time. It took longer for Release 3 to stabilize, probably because there is less total data.

Test Week	Execution Hours	Percent Execution Hours	No. of Defects	Predicted Total No. of Defects	Predicted Residual Defects
10	5,823	58%	75	98	23
11	6,539	65%	81	107	26
12	7,083	71%	86	116	30
13	7,487	75%	90	123	33
14	7,846	78%	93	129	36
15	8,205	82%	96	129	33
16	8,564	86%	98	134	36
17	8,923	89%	99	139	40
18	9,282	93%	100	138	38
19	9,641	96%	100	135	35
20	10,000	100%	100	133	33

Table 3-2. Release 1 Results

Test Week	Execution Hours	Percent Execution Hours	No. of Defects	Predicted Total No. of Defects	Predicted Residual Defects
10	6,634	65%	89	203	114
11	7,229	70%	95	192	97
12	8,072	79%	100	179	79
13	8,484	83%	104	178	74
14	8,847	86%	110	184	74
15	9,253	90%	112	184	72
16	9,712	95%	114	183	69
17	10,083	98%	117	182	65
18	10,174	99%	118	183	65
19	10,272	100%	120	184	64

Table 3-3. Release 2 Results

Test Week	Execution Hours	Percent Execution Hours	No. of Defects	Predicted Total No. of Defects	Predicted Residual Defects
8	3,574	71%	54	163	109
9	4,234	84%	57	107	50
10	4,680	93%	59	93	34
11	4,955	98%	60	87	27
12	5,053	100%	61	84	23

Table 3-4. Release 3 Results

Test Week	Execution Hours	Percent Execution Hours	No. of Defects	Predicted Total No. of Defects	Predicted Residual Defects
10	6,003	53%	29	84	55
11	7,621	67%	32	53	21
12	8,783	78%	32	44	12
13	9,604	85%	36	45	9
14	10,064	89%	38	46	8
15	10,560	93%	39	48	9
16	11,008	97%	39	48	9
17	11,237	99%	41	50	9
18	11,243	99%	42	51	9
19	11,305	100%	42	52	10

Table 3-5. Release 4 Results

Tables 3-2 through 3-5 demonstrate that the predicted total number of defects becomes stable for the simple exponential model, which is the first criterion for a useful model. The second criterion is that the predicted residual defects reasonably approximate field use. Table 3-6 compares the predicted residual defects with the first year of field experience. All of the predictions are surprisingly close to field experience and well within the confidence limits except for Release 2. A two-stage model, combining Releases 2 and 3, did a better job of predicting residual defects and is described later in this section. One criticism of the results in Table 3-6 might be that we had to modify the simple model to obtain them, i.e., the two-stage model for Releases 2 and 3 and the additional test hours from parallel test groups for Release 4. However, these modifications were made as the models were being developed because the differences among releases was evident during the QA test phase rather than in hindsight. Having settled on the basic model structure, it was easy to make these types of model modifications.

Release	Predicted Residual Defects	Defects in First Year
1	33	34
2	64	8
3	23	20
4	10	9
2/3	33	28

Table 3-6. Model Predictions vs. Field Experience

The defects in Table 3-6 include defects found by customers and defects found through internal usage as long as the defects found internally were not part of the next major QA test cycle. The defects found by customers tend to be configurations that are difficult to replicate in QA, e.g., a very large system running continuously for months. The third column in Table 3-6 includes known defects and TPRs that were still open for analysis at the end of the first year. Additional defect data gathered for some releases shows that the number of defects found after the first year is balanced by the number of open TPRs that turn out to be rediscoveries or non-defects. Therefore, the number of defects for the first year shown in Table 3-6 is expected to be close to the total number of defects that will be attributed to that release.

Two-Stage Model Results

Since Release 2 greatly overestimated the number of residual defects, we examined the details of this release. Release 2 was a preliminary release used by very few customers. Release 3 was very similar to Release 2 with some functionality and performance enhancements, and the Release 2 and Release 3 testing overlapped (Release 2 test week 17 was the same as Release 3 test week 1). Therefore, Release 2 and 3 can really be treated as a single release that was tested from Release 2 RQA until Release 3 QAOK in which Release 3 RQA corresponds to the release of additional functionality into the test process. This is the classic setup for the two-stage model described in Section 2.2. Figure 3-2 shows what the data looks like for the two-stage model. This figure shows that the data has the shape of a two-stage model shown in Figure 2-3. Note that the data has an inflection point at about 9,700 hours, which was Release 3 RQA. When we evaluate this data using the two-stage model techniques described in Section 2.2, the predicted total number of defects is 214. From Table 3-1, the total defects in Releases 2 and 3 is 181, so the predicted number of residual defects is 33. From Table 3-6, there were 28 defects in the first year for Releases 2 and 3 combined, which compares favorably with the prediction of 33.

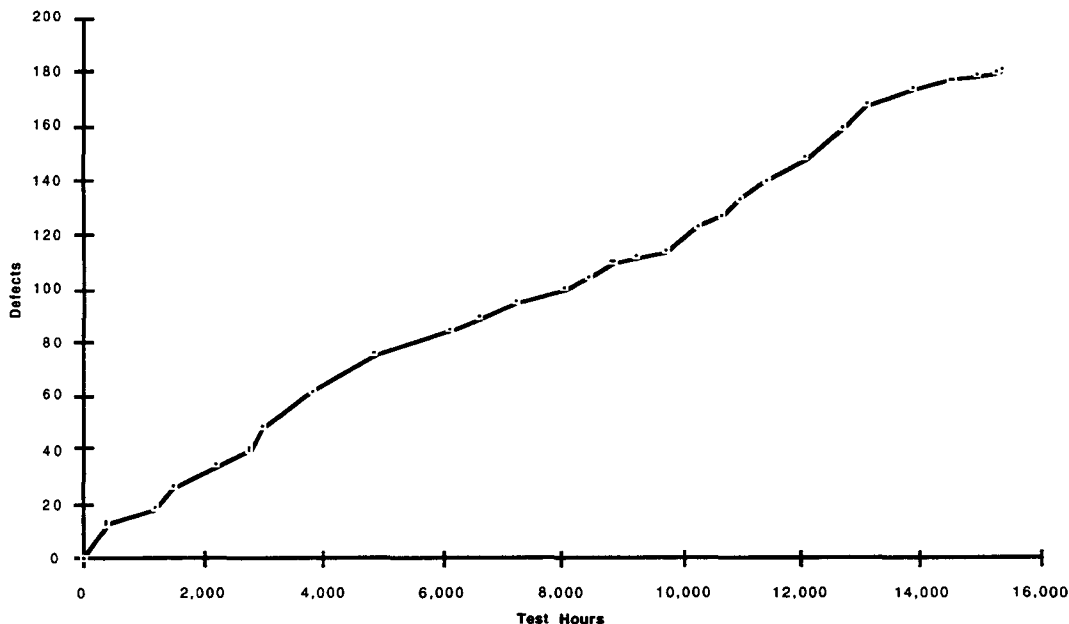


Figure 3-2. Combined Data for Releases 2 and 3

3.3 Results for Different Representations of Test Time

All previously presented results have been calculated using execution time to represent amount of testing rather than calendar time or number of test cases. The reason for this is that our results using calendar time and number of test cases have been poor. Tables 3-2 through 3-5 show that execution time does not correlate well to calendar time, meaning that the testing effort is not spread uniformly throughout the test period. There are times when major defects or schedule conflicts may prevent test execution. Calendar time accumulates during these periods while execution time does not, which is one reason that calendar time

models do not seem to produce credible results. Table 3-7 shows the results of fitting the Release 4 defects to calendar time. We were unable to get a result until week 15 because the curve fit did not converge. After week 15, the prediction was very unstable, especially in comparison to the very stable execution time results, as can be seen from Table 3-7. Similar results with the other releases indicates that execution time is a much better measure of the amount of testing than calendar time in our environment.

Test Week	Execution Hours	Percent Execution Hours	No. of Defects	Predicted Total No. of Defects - Execution Time	Predicted Total No. of Defects - Calendar Time
10	6,003	53%	29	84	
11	7,621	67%	32	53	
12	8,783	78%	32	44	
13	9,604	85%	36	45	
14	10,064	89%	38	46	No Prediction
15	10,560	93%	39	48	457
16	11,008	97%	39	48	178
17	11,237	99%	41	50	125
18	11,243	99%	42	51	101
19	11,305	100%	42	52	85

Table 3-7. Release 4 Results for Calendar Time

We also had poor results using number of test cases to represent amount of time. Table 3-8 shows the test case data and results for Release 3. The total number of test cases has been translated to 10,000. Note that the number of test cases increases faster than the execution hours. This occurs because many simple automated tests that do not take much execution time are run early in the test phase. Again, the prediction was unstable and did not match the field results. Similar results with the other releases indicates that execution time is a much better measure of the amount of testing than number of test cases in our environment.

Test Week	Execution Hours	Percent Execution Hours	No. of Test Cases	Percent Test Cases	No. of Defects	Predicted Total No. of Defects-Execution Time	Predicted Total No. of Defects-Test Cases
1	162	3%	671	7%	6		
2	499	10%	1,920	19%	9		
3	715	14%	2,150	22%	13		
4	1,137	23%	3,112	31%	20		
5	1,799	36%	3,802	38%	28		
6	2,438	48%	5,009	50%	40		
7	2,818	56%	6,443	64%	48		
8	3,574	71%	7,630	76%	54	163	No Prediction
9	4,234	84%	9,263	93%	57	107	204
10	4,680	93%	9,690	97%	59	93	152
11	4,955	98%	9,934	99%	60	87	137
12	5,053	100%	10,000	100%	61	84	132

Table 3-8. Release 3 Results for Number of Test Cases

3.4 Results From Modeling Problem Reports

Our results from using problem reports instead of defects showed that problem reports are an excellent surrogate for defects. These results are shown in Tables 3-9 and 3-10 for Releases 2 and 3.

Test Week	No. of TPRs	Predicted Total No. of TPRs	No. of Defects	Predicted Total No. of Defects	Predicted Total No. of Defects Based on TPRs
1	19		13		
2	31		18		
3	49		26		
4	62		34		
5	71		40		
6	84		48		
7	101		61		
8	123	315	75		193
9	142	289	84		177
10	151	288	89	203	177
11	159	284	95	192	174
12	169	278	100	179	170
13	175	279	104	178	171
14	183	285	110	184	175
15	185	285	112	184	175
16	188	282	114	183	173
17	191	278	117	182	171
18	193	278	118	183	170
19	195	278	120	184	171

Table 3-9. Release 2 Results for TPRs

Test Week	No. of TPRs	Predicted Total No. of TPRs	No. of Defects	Predicted Total No. of Defects	Predicted Total No. of Defects Based on TPRs
1	8		6		
2	12		9		
3	22		13		
4	35		20		
5	47		28		
6	62		40		
7	75		48		
8	84	213	54	163	127
9	89	159	57	107	95
10	94	143	59	93	86
11	100	145	60	87	87
12	101	147	61	84	88

Table 3-10. Release 3 Results for TPRs

The predictions based on TPRs become stable earlier than predictions based on defects because there is more data. We used the ratio of TPRs to defects to predict a total number of defects from the TPR model. This ratio is usually about 60% (recall that the other 40% of TPRs are mainly rediscoveries caused by parallel usage of the products under test). As an example, for Release 2 the ratio of defects to TPRs was 120/195 = 62%. The predicted number of TPRs is 278 at Week 19 of Release 2 testing. Taking 62% of 278 yields 171, which is reasonably close to the final prediction of 184 from the defect model.

During system test, we would use the results from a few weeks preceding the current test week to predict a ratio of defects to TPRs and then use this ratio and the current test week TPR prediction to predict expected defects. The results for Release 3 show that, despite a change in the defect to TPR ratio from 64% in Week 9 to 60% in Week 12, this technique still provides a reasonable prediction of residual defects.

3.5 Results for Different Models

We fit all the different software reliability growth models described in Section 2.3 to the data shown in Table 3-1. The results for Release 1 are shown in Table 3-11. The numbers in the table show the predicted number of total defects for each model at various times in the test process. Note that most of the models become reasonably stable at about the same time as the G-O model but that their predictions of the total number of defects are significantly different than the G-O model. The S-shaped models (G-O S-shaped, Gompertz, Yamada Raleigh) all tended to under predict the total defects. This is expected since the data has the shape of a concave model rather than an S-shaped model. The other concave models (Pareto, Yamada Exponential) all tended to over predict the number of total defects. The models that are variants of the G-O model (Hossain-Dahiya/G-O and Weibull) both predicted exactly the same parameters as the G-O model. The Log-Poisson model is an infinite failure model and does not have a parameter that predicts that total number of defects. To estimate the total number of defects from this model, we estimated the time at which the G-O model would have found 90% of the residual defects and then determined the number of failures that the Log-Poisson model would have predicted at that point in time. The relatively good performance of the Log-Poisson model may be the result of this artificial total defect estimation technique. Our conclusion from these results is that the G-O model was significantly better for our data than the other models.

	Total Defects predicted several weeks after RQA				
Model Name	10 Weeks	12 Weeks	14 Weeks	17 Weeks	20 Weeks
Goel-Oku moto (G-O)	98	116	129	139	133
G-O S-Shaped	71	82	91	99	102
Gompertz	96	110	107	114	112
Yamada Raleigh	77	89	98	107	111
Pareto	757	833	735	631	462
Yamada Exponential	152	181	204	220	213
Hossain-Dahiya/G-O	All results same as G-O model				
Weibull	All results same as G-O model				
Log Poisson	140	153	161	166	160

There were 134 total defects found for Release 1, 100 in QA test, 34 after QA test

Table 3-11. Release 1 Results for Various Models

3.6 Different Correlation Techniques

Throughout this paper we have presented results obtained using the alternative least squares technique described in Section 2.3.3. Table 3-12 shows the results obtained with the other two statistical techniques for all the releases. For Release 1, the alternative least squares technique is more stable than the other two techniques. The standard least squares technique requires that the number of defects change each week because the weekly change is used as the denominator of an equation, so we were unable to solve for the parameters using this technique in weeks 19 and 20. For Releases 2 and 4, the alternative least squares technique appears to be slightly more stable than the maximum likelihood technique. For Release 3, the maximum likelihood technique appears to be slightly more stable than the alternative least squares technique. However, the differences between these two techniques do not appear to be significant. The standard least squares technique appears to be very unstable in some cases, e.g., weeks 16-18 of Release 1, week 19 of Release 2, and week 18 of Release 4. Since the alternative least squares technique is the easiest to use, is slightly more stable, and correlates slightly better to the results from field data, it is our preferred technique.

		Release 1			Release 2				Release 3				Release 4			
Test Wk	Def-ects	ML	LS	LS*	Def-ects	ML	LS	LS*	Def-ects	MLE	LS	LS*	Def-ects	ML	LS	LS*
1	16				13				6				1			
2	24				18				9				3			
3	27				26				13				8			
4	33				34				20				9			
5	41				40				28				11			
6	49				48				40				16			
7	54				61				48				19			
8	58				75				54	124	163	142	25			
9	69				84				57	86	107	87	27			
10	75	113	98	139	89	158	203	163	59	79	93	76	29	65	84	85
11	81	122	107	149	95	162	192	164	60	76	87	72	32	43	53	49
12	86	134	116	169	100	153	179	152	61	78	84	78	32	38	44	36
13	90	144	123	188	104	166	178	170					36	46	45	48
14	93	146	129	183	110	191	184	206					38	51	46	57
15	96	148	129	181	112	179	184	176					39	54	48	61
16	98	150	134	182	114	172	183	165					39	52	48	55
17	99	142	139	148	117	174	182	168					41	57	50	66
18	100	133	138	126	118	179	183	188					42	63	51	325
19	100	126	135	**	120	188	184	217					42	62	52	**
20	100	122	133	**												

* Classical LS technique

** Couldn't solve because the number of defects didn't change from previous week

Table 3-12. Statistical Technique Comparison

Table 3-13 shows the confidence interval results for Release 4. As discussed in Section 2.3.5, the maximum likelihood confidence intervals are asymmetric while the least squares confidence intervals are symmetric. Unfortunately, the maximum likelihood confidence intervals are very wide. The confidence interval range is more than three times larger the predicted residual defects (range at week 19 is 106-51=65 and predicted residual defects are 62-42=20). The classical least squares lower confidence limit is less than the defects experienced, which obviously cannot be true. The confidence intervals derived from the alternative least squares technique are very small - the confidence intervals for weeks 12-14 do not even include the final total defect point estimate. Since these did not seem credible, a second technique based on the Poisson distribution (described in Appendix 1) was used to determine confidence intervals. These confidence seem a little more reasonable but have the same problem of the lower confidence limit being less than the defects experienced. None of these confidence intervals seems very satisfactory although the maximum likelihood confidence intervals are the most credible.

		ML Results			Classical LS Results			Alternative LS Results**				
Test Week	No. of Defects	Total Defects	Lower 5% CL	Upper 95%CL	Total Defects	Lower 5% CL	Upper 95%CL	Total Defects	Lower 5% CL		Upper 95%CL	
11	32	43	36	68	49	20	78	53	46	41	60	65
12	32	38	35	46	36	31	42	44	39	33	50	55
13	36	46	40	65	48	26	70	45	40	34	50	56
14	38	51	44	78	57	20	94	46	42	35	51	57
15	39	54	46	82	61	21	100	48	43	36	52	59
16	39	52	45	74	55	32	78	48	44	37	53	60
17	41	57	48	88	66	22	110	50	46	38	54	61
18	42	63	51	111	325	-3,196	3,846	51	47	39	55	63
19	42	62	51	106	*			52	48	40	56	64

*Couldn't solve because the number of defects didn't change from previous week
**First set of confidence limits, based on t distribution, from Equation (5) in Section 2.3.2. Second set, based on Poisson distribution, from Equation (A6) in Appendix 1.

Table 3-13. Confidence Interval Comparison for Release 4

3.7 Grouped Data Stability

As mentioned in Section 2.1.3, we have access only to weekly data rather than exact defect detection data. To simulate exact data, we took the data for Release 4 and distributed the defects throughout the week in which they arrived. We did this randomly (using the random number generator in Excel) and using a fixed pattern, e.g., 2 failures in a week were assumed to be evenly spaced - at 2.3 and 4.7 days. Table 3-14 shows the results. We thought that having this extra data might cause the prediction to stabilize sooner, but that was not the case. Note that the predictions from the simulated exact data and the weekly grouped data are essentially identical. Our conclusion is that the weekly grouped data works fine for our development process. This is useful because it means that we do not have to try to change the data input and collection processes.

Test Week	Execution Hours	No. of Defects	Predicted Total No. of Defects - Grouped Data	Predicted Total No. of Defects - Ungrouped Data (Random)	Predicted Total No. of Defects - Ungrouped Data (Fixed)
10	6,003	29	84	149	116
11	7,621	32	53	60	61
12	8,783	32	44	57	54
13	9,604	36	45	46	46
14	10,064	38	46	47	47
15	10,560	39	48	47	48
16	11,008	39	48	48	48
17	11,237	41	50	49	49
18	11,243	42	51	49	49
19	11,305	42	52	50	50

Table 3-14. Release 4 Results for Ungrouped Data

3.8 Rerun Test Hours

The test hours that have been used in all the models include test hours for tests that were run a first time and test hours for tests that were rerun. Tests are rerun to either ensure that a defect has been correctly repaired or that fixing a defect did not cause another failure. One hypothesis is that the rerun test hours are less likely to find defects than tests run for the first time because the tests have once been successfully (usually) executed. We tested this hypothesis by counting rerun test hours at only 50% of the value of the initial test execution time. We fit the G-O model to the adjusted hours for Release 1 with the results shown in Table 3-15. The results do not indicate that this technique is any better than our usual technique for fitting the data. We also tried different factors such as 25% for adjusting the data but did not get any better results than we did with the original data. This, together with our excellent results from treating all execution hours as equivalent, leads us to conclude that there is no need to adjust the test hours for modeling our data.

Test Week	No. of Defects	Predicted Total No. of Defects - Rerun hours discounted by 50%	Predicted Total No. of Defects - Rerun Test Hours Not Discounted
10	75	97	98
12	86	116	116
14	93	134	129
17	99	157	139
20	100	159	133

Table 3-15. Release 1 Results for Discounting Rerun Test Hours

Acknowledgment

I would like to thank Jocelyn Miyoshi and Helen Cheung for their help in acquiring and analyzing the data. I would also like to thank Jeff Terry for many enlightening conversations about the Tandem software development and QA processes.

References

1. [Ehrlich,90] Ehrlich, Willa, S. Keith Lee, and Rex Molisani, "Applying Reliability Measurement: A Case Study". *IEEE Software*, March 1990, pp 56-64.

2. [Goel,79] Goel, A. L. and Kazuhira Okumoto, "A Time Dependent Error Detection Model for Software Reliability and Other Performance Measures", *IEEE Trans. Reliability*, vol R-28, Aug 1979, pp 206-211.

3. [Hossain,93] Hossain, Syed and Ram Dahiya, "Estimating the Parameters of a Non-Homogeneous Poisson-Process Model of Software Reliability", *IEEE Trans. Reliability*, vol 42, No. 4, Dec 1993, pp 604-612.

4. [Kececioglu,91] Kececioglu, Dimitri, Reliability Engineering Handbook, Volume 2, Prentice-Hall, 1991.

5. [Lee,93] Lee, Inhwan and Ravishankar K. Iyer, "Faults, Symptoms, and Software Fault Tolerance in the Tandem GUARDIAN90 Operating System", Proceedings of the 23rd International Symposium on Fault-Tolerant Computing (FTCS-23), Toulouse, France, June 22-24, 1993.

6. [Littlewood,81] Littlewood, B., "Stochastic Reliability Growth: A Model for Fault Removal in Computer Programs and Hardware Design", *IEEE Trans. Reliability*, vol R-30, Dec 1981, pp 313-320.

7. [Mood,74] Mood, Alexander, Franklin Graybill, and Duane Boes, Introduction to the Theory of Statistics, McGraw-Hill, 1974.

8. [Musa,87] Musa, John, Anthony Iannino, and Kazuhira Okumoto, Software Reliability, McGraw-Hill, 1987.

9. [Yamada,83] Yamada, Shigeru, Mitsuru Ohba, and Shunji Osaki, "S-Shaped Reliability Growth Modeling for Software Error Detection", *IEEE Trans. Reliability*, vol R-32, Dec 1983, pp 475-484.

10. [Yamada,86] Yamada, Shigeru, Hiroshi Ohtera, and Hiroyuki Narihisa, "Software Reliability Growth Models with Testing Effort", *IEEE Trans. Reliability*, vol R-35, No. 1, April 1986, pp 19-23.

Appendix 1 Least Squares Calculations

An alternative to classical least squares is to minimize the likelihood function directly rather than the log-likelihood function. In this case, we assume that the difference between the cumulative number of defects in week *i* and the model prediction for week *i* is a normally distributed random variable with mean 0. In other words,

$f_i = \mu(t_i) + \varepsilon_i$, where

ε_i are independent, identically distributed, normal random errors with mean 0 and common variance σ^2.

The least squares technique is to minimize the sum of the squared errors, i.e., minimize:

$\Sigma_{i=1}^{w} \varepsilon_i^2 = \Sigma_{i=1}^{w} (f_i - \mu(t_i))^2$, which is Equation (6) in Section 2.3.3, with
w = current number of weeks of QA test
t_i = cumulative test time at the end of the ith week
f_i = cumulative number of failures at the end of the ith week.

For the G-O model, Equation (6) becomes:

(A1) $\Sigma_{i=1}^{w} (f_i - a(1-e^{-bt_i}))^2$, which is Equation (7) in Section 2.3.3.

The confidence interval for *a* assuming that *a* is normally distributed is

(A2) $a \pm t_{w-2,\alpha/2}\,(\mathrm{Var}(a))^{0.5}$, which is Equation (5) in Section 2.3.2.

The calculation of the variance of *a* is described in [Musa,87,Section 12.3] for the classical least squares technique described in Section 2.3.2. In this Appendix, the calculation of the variance of *a* for the alternative least squares technique is derived.

The assumptions that the errors ε_i are normally distributed with a common variance σ^2 means that the following equation can be used for the variance of *a* and *b* (see e.g. [Musa,87,Equation 12.150]):

(A3) $\mathrm{Var}(a) = (\sigma^2/SS)\, \Sigma_{i=1}^{w} [(\delta/\delta b)\,(a(1-e^{-bt_i}))]^2$, where
$\sigma^2 = \Sigma_{i=1}^{w} (f_i - a(1-e^{-bt_i}))^2/(w-2)$
$SS = a^2\, \Sigma_{i=1}^{w} (1-e^{-bt_i})^2\, \Sigma_{i=1}^{w} t_i^2 (1-e^{-bt_i})^2 - a^2\, \{\Sigma_{i=1}^{w} t_i (1-e^{-bt_i})^2\}^2$
$(\delta/\delta b)$ is a partial derivative with respect to *b*, so $(\delta/\delta b)\,(a(1-e^{-bt_i}) = at_i e^{-bt_i}$.

(A4) $\mathrm{Var}(b) = (\sigma^2/SS)\, \Sigma_{i=1}^{w} [(\delta/\delta a)\,(a(1-e^{-bt_i}))]^2$, where
$(\delta/\delta a)$ is a partial derivative with respect to *a*, so $(\delta/\delta a)\,(a(1-e^{-bt_i}) = 1-e^{-bt_i}$.

To find a confidence interval for *a*, Equation (A2) can be used directly by assuming that *a* is normally distributed and substituting from Equation (A3). This method was used to derive the first set of confidence intervals reported in Section 3.6. Alternately, Equation (A4) can

be used to derive a confidence interval for *a* assuming that *b* is normally distributed, and we can find a confidence interval for *a* by substituting the confidence limits for *b* into:

(A5) $a = (f_w - \varepsilon_w)/(1 - e^{-bt_w})$.

A different method for calculating confidence intervals is to assume that f_i is a Poisson random variable with expected value $\mu(t_i)$. Then, from P.270 of [Musa,87], the α per cent confidence interval (e.g., 95%) for f_i is given by:

$\mu(t_i) \pm Z_{1-\alpha/2}\, \mu(t_i)^{0.5}$, which, since $\mu(\infty) = a$, becomes:

(A6) $a \pm Z_{1-\alpha/2}\, a^{0.5}$.

Equation (A6) was used to derive the second set of confidence intervals reported in Section 3.6. This is not a very satisfying form for a confidence interval because it depends only on the parameter and not on the data used to derive the parameter.

Appendix 2 Parameter Scaling

In this appendix, we show that the parameter estimates scale linearly for all statistical techniques. Also, we show that the confidence intervals for the least squares method scale linearly while the confidence intervals for the maximum likelihood method do not. Linear parameter scaling means that if we multiply the number of defects observed by a factor k_1, then the estimate of the total number of defects will also be scaled by a parameter k_1. It is very important that the parameter estimates scale linearly since the data has been scaled. If they did not, then the data translation technique used throughout this report would not be useful because other researchers could not duplicate the results. It is also important that test time scaling does not affect the predicted total number of defects, i.e., if we scale the test time by a factor k_2, then the estimate of the total number of defects will be unchanged. Confidence interval scaling means that the confidence intervals should scale by the same factor as the total number of defects. For example if the estimated total defects parameter was 100 with upper and lower confidence intervals of 80 and 120 and we multiplied the number of defects observed in each test interval by 2, then the scaled total defects parameter should be 200 and the scaled confidence intervals should be 160 to 240. It is less important, but unfortunate, that the confidence intervals do not scale linearly for the maximum likelihood technique.

For the maximum likelihood technique, the following equation must be solved (see Section 2.3.1):

(A7) $\Sigma_{i=1}^{w}(f_i - f_{i-1})(t_i e^{-bt_i} - t_{i-1}e^{-bt_{i-1}})/(e^{-bt_i} - e^{-bt_{i-1}}) = f_w t_w/(1 - e^{bt_w})$, where

w = current number of weeks of QA test

t_i = cumulative test time at the end of the ith week

f_i = cumulative number of failures at the end of the ith week.

If we replace the number of failures f_i by $k_1 f_i$ and replace the test time t_i by $k_2 t_i$, the same solution is obtained for Equation (A7) if the parameter *b* is replaced by b/k_2. From Equation (3) in Section 2.3.1, the total defects parameter is then

$k_1 f_w/(1 - e^{-(b/k_2)(k_2 t_w)}) = k_1 f_w/(1 - e^{-bt_w}) = k_1 a$,

where *a* is the total defects parameter obtained prior to scaling the data. Thus, the total defects parameter scales linearly.

From Equation (2) in Section 2.3.1, the α per cent confidence interval (e.g., 95%) for *b* is given by:

(A8) $b \pm Z_{1-\alpha/2}/(I_0(b))^{0.5}$, where

$$I_0(b) = \Sigma_{i=1}^{w}(f_i - f_{i-1})(t_i - t_{i-1})^2 e^{-b(t_i + t_{i-1})}/(e^{-bt_{i-1}} - e^{-bt_i})^2 - f_w t_w^2 e^{bt_w}/(e^{bt_w} - 1)^2$$

If we replace the number of failures f_i by $k_1 f_i$ and replace the test time t_i by $k_2 t_i$ in the equation for $I_0(b)$, the result is $k_1 k_2^2 I_0(b)$. If we then substitute into Equation (A8), the result is

$b/k_2 \pm Z_{1-\alpha/2}/(k_1 k_2^2 I_0(b))^{0.5} = (1/k_2)[b \pm Z_{1-\alpha/2}/(k_1 I_0(b))^{0.5}]$.

Note that the extra factor of k_1 in the denominator prevents the confidence interval for *b* from scaling linearly. Then, since the confidence interval for the parameter *a* is also derived from the parameter *b*, the confidence interval for parameter *a* does not scale linearly.

For the classical least squares technique, the following equation must be solved for the G-O model (see Section 2.3.2):

(A9) $\Sigma_{i=1}^{w}(\ln((f_i - f_{i-1})/(t_i - t_{i-1})) - \ln(b) - \ln(a - f_i))^2$

If we replace the number of failures f_i by $k_1 f_i$ and replace the test time t_i by $k_2 t_i$ in Equation (A9), the result is

(A10) $\Sigma_{i=1}^{w}(\ln((f_i - f_{i-1})/(t_i - t_{i-1})) + \ln(k_1) - \ln(k_2) - \ln(b) - \ln(a - k_1 f_i))^2$

If we replace *a* with $k_1 a$ and *b* with b/k_2, we get the Equation (A9) with the same minimum. Thus, a scales linearly. The confidence intervals for *a* are:

$$a \pm t_{w-2,\alpha/2}\,(\mathrm{var}(a))^{0.5}$$

Since $\mathrm{var}(k_1 a) = k_1^2 \mathrm{var}(a)$, the confidence intervals scale linearly.

For the alternative least squares technique, the equation to minimize is:

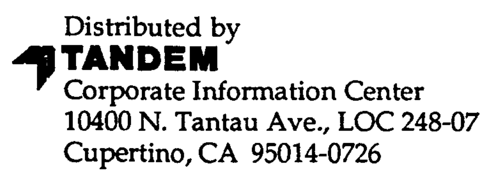
(A11) $\Sigma_{i=1}^{w}(f_i - \mu(t_i))^2$

$= \Sigma_{i=1}^{w}(f_i - a(1 - e^{-bt_i}))^2$ for the G-O model

If we replace the number of failures f_i by $k_1 f_i$ and replace the test time t_i by $k_2 t_i$ in Equation (A11), the result is

(A12) $\Sigma_{i=1}^{w}(k_1 f_i - a(1 - e^{-k_2 b t_i}))^2$

If we replace *a* with $k_1 a$ and *b* with b/k_2 in Equation (A12), we get an equation with the same minimum as Equation (A11). Thus, *a* scales linearly. The first method of finding confidence intervals for the alternative least squares technique is the same as the classical least squares technique. As shown above, these confidence intervals scale linearly. The second technique defines the confidence intervals for *a* as $a \pm Z_{1-\alpha/2}\,(a)^{0.5}$. If we replace *a* with $k_1 a$, the confidence intervals are $k_1 a \pm Z_{1-\alpha/2}\, k_1^{0.5} a^{0.5}$. Therefore, the confidence interval for parameter *a* does not scale linearly.



Distributed by
TANDEM
Corporate Information Center
10400 N. Tantau Ave., LOC 248-07
Cupertino, CA 95014-0726