7.0 PREDICTION

Reliability prediction is useful in a number of ways. A prediction methodology provides a uniform, reproducible basis for evaluating potential reliability during the early stages of a project. Predictions assist in evaluating the feasibility of proposed reliability requirements and provide a rational basis for design and allocation decisions.

Predictions that fall short of requirements at any level signal the need for both management and technical attention. In some cases a shortfall in reliability may be offset by the use of fault tolerance techniques. For hardware, adding redundancy will often result in increased reliability. Software reliability may be improved by a focused inspection, defect removal and test effort. Software fault tolerance techniques, such as N-version programming and recovery blocks, are used as a last resort because of the high cost and controversial impact on reliability.

Hardware reliability prediction provides a constant failure rate value for the "inherent reliability" of the product, the estimated reliability attainable when all design and production problems have been worked out. A hardware reliability growth model is used to monitor product reliability in the period during which the observed reliability advances toward the inherent reliability.

Software reliability prediction provides a projection of the software failure rate at the start of or any point throughout system test. A software reliability growth model covers the period after the prediction, where reliability improves as the result of testing and fault correction.

Hardware and software reliability predictions, when adjusted by their respective growth models to coincide with the same point in time, can be combined to obtain a prediction of the overall system reliability.

Table 7-1 (page 7-3) lists the software reliability prediction procedures to use during each software development life cycle phase. When system test begins, actual failure data can be used to statistically estimate the growth model parameters (see Section 8).

7.1 Hardware Reliability Prediction.

Hardware reliability prediction is a process of quantitatively assessing an equipment design. Techniques have been established so that hardware reliability predictions may be applied and interpreted uniformly. The final outcome of a prediction is a constant failure rate that can be combined with other failure rates in a system model.

7.2 Software Reliability Prediction.

Metrics are used to predict a variety of measures including the initial failure rate λ_0, final failure rate, fault density per executable lines of code, fault profile, as well as the parameters of a software reliability growth model. The final outcomes of a software reliability prediction include:

- Relative measures for practical use and management.
- A prediction of the number of faults expected during each phase of the life cycle.
- A constant failure rate prediction at system release that can be combined with other failure rates.

The major difference between software reliability prediction and software reliability estimation is that predictions are performed based on historical data while estimations are based on collected data. Predictions, by there nature, will almost certainly be less accurate than estimations. However, they are useful for improving the software reliability during the development process. If the organization waits until collected data is available (normally during testing), it will generally be too late to make substantial improvements in software reliability. The predictions should be performed iteratively during each phase of the life cycle and as collected data becomes available the predictions should be refined to represent the software product at hand.

A software reliability prediction is performed early in the software life cycle, but the prediction provides an indication of what the expected reliability of the software will be either at the start of system test or the delivery date. It is largely based on the projected fault count at the point system test is initiated.

While hardware analysts will perform predictions to determine what improvements, if any, can be made in designing and selecting parts, the software analysts will perform predictions to determine what improvements, if any, can be made to the software development techniques employed and the rigor with which the process is carried out. The techniques can be on a global level, such as organization procedures, or they can be on a local level such as the complexity of each software unit. The software analyst, like the hardware analyst, must be involved in the software engineering day-to-day activities to be able to measure the software reliability parameters and to be able to understand what improvements can be made.

One important benefit from performing predictions is to correlate the software methods and techniques employed to the actual failure rate later experienced. This comparison can lead to improved software methods and techniques, particularly testing techniques.

Tables 7-1 lists five software reliability prediction techniques that are available. Table 7-2 lists the phases where the methods are most applicable.

TABLE 7-1. Software Reliability Prediction Techniques

Section	Prediction Method	Capabilities	Description of outputs
7.2.1	Rome Laboratory TR-92-52 Software Reliability Measurement and Test Integration Techniques	Allows for tradeoffs.	Produces a prediction in terms of fault density or estimated number of inherent faults.
7.2.2	Raleigh Method	The profile of predicted faults over time and not just the total number is needed. Can be used with the other prediction models.	Produces a prediction in the form of a predicted fault profile over the life of the project.
7.2.3	Industry data collection	Applicable for any industry.	Produces a prediction of fault density per function points based on historical data collected in industry.
7.2.4	Musa's Model	Predicts failure rate at start of system test that can be used later in reliability growth models.	Produces a prediction of the failure rate at the start of systems test.
7.2.5	Historical data collection	Can be most accurate, if there is organization wide commitment.	Produces a prediction of the failure rate of delivered software based on company wide historical data.

TABLE 7-2. Prediction Techniques by Phase

Phase	Procedure
Proposal and Pre-contractual	7.2.1.1, 7.2.2, 7.2.3, 7.2.4, 7.2.5
Requirements Analysis	7.2.1.1, 7.2.2, 7.2.3, 7.2.4, 7.2.5
Preliminary Design	7.2.1.2, 7.2.2, 7.2.3, 7.2.4, 7.2.5
Detailed Design	7.2.1.2, 7.2.2, 7.2.3, 7.2.4, 7.2.5
Coding and CSU Testing	7.2.1.3, 7.2.2, 7.2.3, 7.2.4, 7.2.5
CSC Integration and Testing	7.2.1.3, 7.2.2, 7.2.3, 7.2.4, 7.2.5

7.2.1 RL-TR-92-52, "Software Reliability Measurement and Test Integration Techniques" Method. RL-TR-92-52 contains empirical data that was collected from a variety of sources, including the Software Engineering Laboratory. There were a total of 33 data sources representing 59 different projects. The model consists of 9 factors that are used to predict the fault density of the software application. The 9 factors are:

TABLE 7-3. Summary of the RL-TR-92-52 Model

Factor	**Measure**	**Range of values**	**Applicable Phase***	**Tradeoff Range**
A - Application	Difficulty in developing various application types	2 to 14 (defects/KSLOC)	A-T	None - fixed
D - Development organization	Development organization, methods, tools, techniques, documentation	.5 to 2.0	If known at A, D-T	The largest range
SA - Software anomaly management	Indication of fault tolerant design	.9 to 1.1	Normally, C-T	Small
ST - Software traceability	Traceability of design and code to requirements	.9 to 1.0	Normally, C-T	Large
SQ - Software quality	Adherence to coding standards	1.0 to 1.1	Normally, C-T	Small
SL - Software language	Normalizes fault density by language type	Not applicable	C-T	N/A
SX - Software complexity	Unit complexity	.8 to 1.5	C-T	Large
SM - Software modularity	Unit size	.9 to 2.0	C-T	Large
SR - Software standards review	Compliance with design rules	.75 to 1.5	C-T	Large

Key A- Concept or Analysis Phase
D- Detailed and Top level Design
C - Coding
T - Testing

*If there are software development policies in place which are defined and it is known what these items will be (even though the code does not exist yet) then these items can be used earlier in the prediction phase. However, the analyst needs to be certain that the prediction reflects what software engineering practices will actually be performed.

There are certain parameters in this prediction model that have tradeoff capability. This means that there is a large difference between the maximum and minimum predicted values for that particular factor. Performing a tradeoff means that the analyst determines where some changes can be made in the software engineering process or product to experience an improved fault density prediction. A tradeoff is valuable only if the analyst has the capability to impact the software development process.

The tradeoff analysis can also be used to perform a cost analysis. For example, a prediction can be performed using a baseline set of development parameters. Then the prediction can be performed again using an aggressive set of development parameters. The difference in the fault density can be measured to determine the payoff in terms of fault density that can be achieved by optimizing the development. A cost analysis can also be performed by multiplying the difference in expected total number of defects by either a relative or fixed cost parameter.

The output of this model is a fault density in terms of faults per KSLOC. This can be used to compute the total estimated number of inherent defects by simply multiplying by the total predicted number of KSLOC. If function points are being used, they can be converted to KSLOC by using Table 7-9. Fault density can also be converted to failure rate by using one of the following:

1) collected test data,
2) historical data from other projects in your organization, and/or
3) the transformation table supplied with the model, shown in Table 7-4.

TABLE 7-4. Transformation Ratio

Application type	**Conversion from fault density to failure rate**
Airborne	6.28
Strategic	1.2
Tactical	13.8
Process control	3.8
Production center	23
Developmental	132.6
Average	10.6

These are listed in the order of preference. Ideally, the developing organization should determine a conversion rate between fault density and failure rate. If that data is not available then this technique supplies a conversion ratio table. This table is based on data generated during the development of the RL-TR-92-52 report. The predicted fault density output from this model can also be used as an input to the Musa prediction model in 7.2.4.

The values of many of the parameters in this model may change as development proceeds. The latest updated values should always be used when making a prediction. The predictions will tend to become more and more accurate as the metrics from each successive phase become available and as the values are updated to more closely reflect the characteristics of the final design and implementation. The details of this model are not contained in this notebook.

7.2.1.1 Proposal, Pre-Contract or Requirements Phase Prediction.
This method requires only that information concerning the type of application and development organization be known. The lower the computed value, the lower the fault density and predicted failure rate.

TABLE 7-5. Proposal/Pre-Contract/Analysis Phase Factors

Factor	Measure	Range of values
A - Application	Difficulty in developing various application types	2 to 14
D - Development organization*	Development organization, methods, tools, techniques, documentation	.5 to 2.0

*May not be known during this phase

Steps.

A. Determine the characteristics of the application type to be developed using the check sheet provided with technical report, RL-TR-92-52.

B. Determine the approximate size of the application in source lines of code and number of units.

C. Determine an initial fault density and estimated number of inherent faults using the checklists included in the technical report.

- Compute D = predicted faults per KSLOC = A if D is not known OR

- Compute FD = predicted faults per KSLOC = A*D if D is known

- N = estimated number inherent faults = FD*KSLOC

During the concept and requirements phases the A factor is always known and the D factor may be known.

D. If necessary, convert the fault density to failure rate using a conversion technique.

E. Perform tradeoffs with the D factor to determine what techniques would be necessary to achieve an objective fault density or failure rate. Re-compute fault density.

7.2.1.2 Design Phase Prediction.

This method requires only that information concerning the type of application, development organization, and design requirements be known. The lower the computed value, the lower the fault density and predicted failure rate.

TABLE 7-6. Design Phase Factors

Factor	**Measure**	**Range of values**
A - Application	Difficulty in developing various application types	2 to 14
D - Development organization	Development organization, methods, tools, techniques, documentation	.5 to 2.0
SA - Software anomaly management*	Indication of fault tolerant design	.9 to 1.1
ST - Software traceability*	Traceability of design and code to requirements	.9 to 1.0
SQ - Software quality*	Adherence to coding standards	1.0 to 1.1

*Even though there is typically no code yet in the design phase, the parameters may be gauged based on coding practices that may be in place.

Steps.

A. Verify that the A factor determined previously in step 7.2.1.1 is still valid. Make any necessary refinements.

B. Determine and/or refine the approximate size of the application in source lines of code and number of units.

C. Determine an initial fault density and estimated number of inherent faults using the checklists included in the technical report.

- Compute FD = predicted faults per lines of code = A*D

- If there are coding policies in place then compute FD = predicted faults per KSLOC = A*D*SA*ST*SQ

- Compute N = estimated number inherent faults = FD*KSLOC

During the design phases the A and D factors should be known.

D. If necessary, convert fault density to failure rate using a conversion factor.

E. Perform tradeoffs with the D factor to determine what techniques would be necessary to achieve an objective fault density or failure rate.

7.2.1.3 Code, Unit Test and Integration Phase Prediction.
This method requires that information concerning the type of application, development organization, design requirements and coding practices be known. The lower the computed value, the lower the fault density and predicted failure rate.

TABLE 7-7. Coding/Unit Testing/Integration Phase Factors

Factor	**Measure**	**Range of values**
A - Application	Difficulty in developing various application types	2 to 14
D - Development organization	Development organization, methods, tools, techniques, documentation	.5 to 2.0
SA - Software anomaly management	Indication of fault tolerant design	.9 to 1.1
ST - Software traceability	Traceability of design and code to requirements	.9 to 1.0
SQ - Software quality	Adherence to coding standards	1.0 to 1.1
SL - Software language	Normalizes fault density by language type	n/a
SX - Software complexity	Unit complexity	.8 to 1.5
SM - Software modularity	Unit size	.9 to 2.0
SR - Software standards review	Compliance with design rules	.75 to 1.5

Steps.

A. Verify that the A factor determined previously in step 7.2.1.1 is still valid. Make any necessary refinements.

B. Refine the calculations for the D factor and the SA factor.

C. Determine and/or refine the approximate size of the application in source lines of code and number of units.

D. Determine an initial fault density and estimated number of inherent faults using the checklists included in the technical report.

- FD = predicted faults per lines of code = A*D*SA*ST*SQ*SL*SX*SM*SR

- N = estimated number inherent faults = FD*KSLOC

E. If necessary convert the fault density to failure rate using a conversion technique.

F. Perform tradeoffs with the applicable D and S factors to determine what techniques would be necessary to achieve an objective fault density or failure rate.

7.2.2 Raleigh Model[1].

This model predicts fault detection over the life of the software development effort and can be used in conjunction with the other prediction techniques in Section 7.2. Software management may use this profile to gauge the defect status. This model assumes that over the life of the project that the faults detected per month will resemble a Raleigh curve (Figure 7-1).

Steps.

A. Obtain the milestones for the schedule, in particular the

- Start date and total months in project
- Date of expected full operational capability - t_d

B. Estimate the number of faults over the life of the project - E_r. The other prediction techniques can be used to predict the fault density. The fault density can then be multiplied by either KSLOC or function points as depending on the prediction technique used.

C. From these unknowns, a Raleigh curve can be calculated by solving for each month t (1 to number of months in project) using this equation , $E_m = (6 * E_r/t_d^2) * t * \exp(-3t^2/t_d^2)$. When finished, the result should be a plot that resembles a Raleigh distribution.

D. Use this profile to gauge the fault detection process during each phase of development. In particular, this profile can be used to gauge the original schedule estimate and the prediction for the

[1] "Measures for Excellence", Larry Putnam, Ware Myers, Quantitative Software Management, Yourdon Press, Englewood Cliffs, NJ, 1992.

total number of defects. For example, the estimated number of defects impacts the height of the curve while the schedule impacts the length of the curve. If the actual defect curve is significantly different from the predicted curve then one or both of these parameters may have been estimated incorrectly and should be brought to the attention of management.

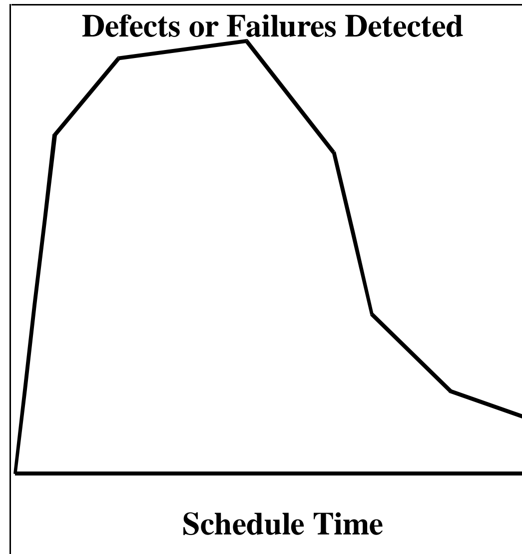

FIGURE 7-1. Raleigh Curve

7.2.3 Industry Data.

Table 7-8 summarizes data that has been collected by industry, in particular by Software Productivity Research, Inc.[2] The output from this prediction technique is defects per function points. This metric can be used to predict the total estimated number of inherent defects and can also be used as an input to the Musa prediction model in 7.2.4. See the Appendix for procedures for calculating function points. The potential defects are those that are discovered at any time during development. The delivered defects are those that are discovered after delivery.

Steps.

A. Compute function point measure for each unit. See instructions in the Appendix.

B. Determine the capability level of the software organization developing the software. Keep in mind that there may be joint efforts between more than one software organization and therefore more than one capability level. If that is the case, then perform a separate prediction for each organization based on the function points developed by each organization.

   If the capability level is not known then determine the industry type that most closely represents this software.

C. Use Table 7-8 as an estimate of the potential defects per function point and the delivered defects per function point.

[2] "Measuring Global Software Quality", Software Productivity Research, Capers Jones, Burlington, MA, 1995.

TABLE 7-8. Industry Data Prediction Technique

CMM Approach	
Measure	**Average defects/ function points**
Typical defect potential and delivered defects for SEI CMM Level 1	5.0 potential .75 delivered
Typical defect potential and delivered defects for SEI CMM Level 2	4.0 potential .44 delivered
Typical defect potential and delivered defects for SEI CMM Level 3	3.0 potential .27 delivered
Typical defect potential and delivered defects for SEI CMM Level 4	2.0 potential .14 delivered
Typical defect potential and delivered defects for SEI CMM Level 5	1.0 potential .05 delivered

Industry Approach	
Measure	**Average defects/ function points**
Delivered defects per industry	System Software - .4 Commercial Software - .5 Information Software - 1.2 Military Software - .3 Overall average - .65

7.2.4 Musa Prediction Method.

This prediction technique is used to predict, prior to system testing, what the failure rate will be at the start of system testing. This prediction can then later be used in the reliability growth modeling. This prediction technique also allows for a prediction in terms of failure rate which can be combined with the hardware failure rate predictions.

At any point, an executing computer program exhibits a constant failure rate λ, provided that the code is *frozen* and the operational profile is stationary. A constant failure rate implies an exponential time-to-failure distribution; therefore, the reliability (probability that the program executes without failure for a period of time t') is given by

$$R(\tau') = \exp[-\lambda\tau'] \quad \textbf{(7.1)}$$

The reliability of a software configuration item will change as the software is tested and repair activity takes place. Consequently, a software reliability prediction must be associated with a particular point in time. The earliest point that it makes sense to *estimate* the reliability of the software is when the software is fully integrated and is executed in an environment that is representative of its operational use. This point is the start of system test and is designated t = 0. For any later point in time, t indicates the cumulative execution time since the start of system test.

The failure rate will vary over time. The failure rate at the instant t is denoted $\lambda(\tau)$. When the program code is unchanging during operation, the software may exhibit a constant failure rate $\lambda = \lambda(\tau)$. The failure rate predicted by this model is the initial failure rate $\lambda_0 = \lambda(0)$, the failure rate the software is expected to exhibit at the beginning of system test ($\tau = 0$. The prediction procedures in this section provide λ_0. The procedures in Section 7.3.4 employ the software reliability growth model to estimate additional quantities, such as the schedule and resource impact to achieve a failure rate objective.

For this prediction method, it is assumed that the only thing known about the hypothetical program is a prediction of its size and the processor speed.

This model assumes that failure rate of the software is a function of the number of faults it contains and the operational profile. The number of faults is determined by multiplying the number of developed executable source instructions by the fault density. *Developed* excludes re-used code that is already debugged. *Executable* excludes data declarations and compiler directives. For fault density at the start of system test, a value for faults per KSLOC needs to be determined. For most projects the value ranges between 1 and 10 faults per KSLOC. Some developments which use rigorous methods and highly advanced software development processes may be able to reduce this value to 0.1 fault/KSLOC.

The measurement of processor speed is complicated by the fact that each instruction takes a different amount of time, depending on the nature of the operation and where the operands reside. A unit such as "million instructions per second" (MIPS) implies an average taken over some arbitrary mix of instructions. The best way to determine the average instruction execution rate, denoted r, is through benchmarking, using an application program and environment representative of the program whose reliability is being predicted. Second best, a "MIPS rating" can be obtained from the computer vendor.

<u>Steps.</u>

A. Determine the processor speed, r, in instructions per second.

B. To compute the number of object instructions I, take the number of executable lines of code and multiply by the code expansion ratio, supplied in Table 7-9[3] (previous page). Use this table only if real project data is not available. The rationale behind this data is that the relationship between a line of code and a machine instruction varies depending on the language. Also, the relationship between a line of code and a function point also vary with language.

---

[3] "“Backfiring” or Converting Lines of Code Metrics Into Function Points", Capers Jones, October 6, 1995, Software Productivity Research, Burlington, MA.

TABLE 7-9. Code Expansion Ratios

Programming Language	Expansion Ratio	Mean Source Statements/Function Point
Basic Assembly	1.0	320
Macro Assembly	1.5	320
C	2.5	128
Interpreted Basic	2.5	128
2nd Generation language	3.0	107
Fortran	3.0	107
ALGOL	3.0	107
COBOL	3	107
CMS2	3	107
JOVIAL	3	107
Pascal	3.5	91
3rd Generation language	4.0	80
PL/I	4.0	80
Modula 2	4.0	80
Ada 83	4.5	71
Prolog	5.0	64
Lisp	5.0	64
Forth	5.0	64
Quick Basic	5.5	58
C++	6.0	53
Ada 9X	6.5	49
Database Default	8.0	40
Visual Basic	10.0	32
APL	10.0	32
SMALLTALK	15.0	21
Generators	20.0	16
Screen Painters	20.0	16
SQL	27.0	12
Spreadsheet Default	50.0	6

C. Estimate the fault content ω by using the prediction techniques in Section 7.2.1, 7.2.3 and 7.2.5. In the event that only function points are known, this metric can be converted using Table 7-9.

D. Calculate the initial failure rate using formula **(6.8),** $\lambda_0 = r_i \bullet K \bullet \omega_{0i} / I_i$ **.**

<u>Example:</u>
A 20,000-line Ada program is to be developed. It will execute on a 2-MIPS machine. Assume six (6) defects per KSLOC. What failure rate can be expected at the beginning of system test?

The number of object instructions is calculated by multiplying the 20,000 executable lines of source code by the code expansion ratio for Ada, 4.5, to yield

$$I = (20{,}000 \text{ source lines}) \left(4.5 \frac{\text{object instructions}}{\text{source line}} \right)$$

$$= 90{,}000 \text{ object instructions}$$

The fault content is predicted as

$$\omega_0 = 6 \bullet \Delta I_S =$$

$$\left(\frac{6 \text{ faults}}{1000 \text{ LOC}} \right) x (20{,}000 \text{ LOC}) = 120 \text{ faults}$$

The initial failure rate is then computed by

$$\lambda_0 = r \bullet \omega_0 \bullet K / I$$

$$= \left(\frac{2{,}000{,}000 \text{ instructions}}{\text{second}} \right)$$

$$x\,(120 \text{ faults})$$

$$x \left(4.20x\,10^{-7} \frac{\text{failures}}{\text{fault}} \right)$$

$$/\,90{,}000 \text{ instructions} = 0.00111888 \text{ failures per second}$$

The prediction technique presented thus far relies on the new program's predicted size and processor speed. Beginning with the requirements analysis phase of software development, product/process metrics become available. These metrics can be used in conjunction with empirically obtained prediction models to provide better predictions. In order to determine the software reliability growth model parameters (see Section 7.3.4), the value of ω_0 needs to be retained for use during later phases. If the projected number of developed source lines of code changes, this value should be updated.

7.2.5 Historical Data Collection.
The software development organization can collect interval as well as industry wide historical data to predict software failure rates. The accuracy of this method is completely dependent on the availability and completeness of the data collected. This method is generally considered to be the most expensive, but from an accuracy perspective is preferred.

The collection, storage and analysis of data about the development of the software products as they correlate to reliability and failure rate can be invaluable in discovering the relationship between the process and the product[4].

7.3 Use of Predictions for Project Planning and Control.
The prediction techniques presented in Section 7.2 can be used for planning and control as described in Sections 7.3.2 through 7.3.4 that follow. There are also other industry metrics used for planning and control, described in Section 7.3.1.

7.3.1 RL-TR-92-52 Model.
This model, discussed in Section 7.2.1 and 7.2.2 and 7.2.3, can be used for planning and control as well as for prediction purposes. This model can be used to obtain relative as well as absolute measures of reliability. For example, the factors in this model that have the widest possible range of values are the following:

- Development factor
- Complexity factor
- Modularity factor
- Software Review factors

These factors provide relative improvement values. They also allow comparisons between projects. Cost comparisons can be performed by assessing the improvement in fault density of a more aggressive development approach.

7.3.2 The Raleigh Model.
The Raleigh method discussed in Section 7.2.2 can be used to gauge the defect discovery process. The height of this curve is based on the estimated number of inherent faults E_r. The width of this curve is based on the accuracy of the milestone scheduling or the effectiveness of the assurance activities at each phase. This curve should be updated in the event that either one of these estimates is updated.

[4] Software Reliability Handbook, Edited by Paul Rook, Centre for Software Reliability, Elsevier Applied Science, London, 1990.

7.3.3 Industry Metrics Used.

Some practical and measurable means of planning and controlling software reliability have been developed in industry[5]. These metrics can be used to gauge the reliability and/or quality of a project. These metrics are shown in Table 7-10.

TABLE 7-10. Using Metrics For Planning and Control

Measure	**Indicator of Good Reliability and/or Quality**	**Average**	**Poor**
Low defect potential (defects detected during development)	<1 defect per function point		
High defect removal efficiency	> 95% of all defects are removed prior to delivery		
Stability of requirements	< 2.5% change in baselined requirements		
Achieving explicit requirements	> 97.5 % explicit requirements verified		
Defects per function point experienced after test	0.06	0.44	0.75
Productivity	39 function points or 4250 SLOC per man year	23 function points or 2500 SLOC per man year	12 function points or 1100 SLOC per man year

[5] "Measuring Global Software Quality", Software Productivity Research, Capers Jones, Burlington, MA, 1995.

TABLE 7-10. Using Metrics For Planning and Control (continued)

Measure	**Indicator of Good Reliability and/or Quality**	**Average**	**Poor**
Best case vs. average defect potentials in terms of function points per phase of life cycle	Reqs - .20 Design - .25 Code - .25 Doc - .20 Bad fix - .1 Total 1.0 defect per function point	Reqs - 1.0 Design - 1.25 Code - 1.25 Doc - 1.0 Bad fix - .5 Total 5.0 defect per function point	
Best case delivered defects per function points.	Reqs - .02 Design - .0125 Code - .003 Doc- .01 Bad fix - .01 Total .0560 delivered defects per function points	Reqs - .16 Design - .10 Code - .024 Doc- .08 Bad fix - .08 Total .444 delivered defects per function points	
Defect Removal Efficiency - Ability to remove defects without introducing new ones during development	CMM Level 5 - 95% CMM Level 4 - 93%	CMM Level 3 - 91% CMM Level 2 - 89%	CMM Level 1-85%
Techniques used to achieve results:	• Formal inspections • Joint Application Design • Quality metrics • Removal efficiency measurement • Functional metrics • Active quality assurance • User satisfaction surveys • Formal test planning • Quality estimation tools • Complexity metrics • Quality Function Deployment		

7.3.4 The Musa Reliability Growth Method.

The failure rate predicted by the prediction technique in Section 7.2.4 is λ_0, the failure rate at the start of system test. To determine the failure rate at any time t into system test, the software reliability growth model (see Section 8) is employed.

The growth model parameters are β and ν_0. The parameter β is the (expected) decrement in failure rate per failure occurrence. The parameter ν_0 is the *total failures*: the number of failures that must be experienced to uncover and remove all faults. They are obtained from the predicted values of the initial failure rate λ_0 and fault content ω_0. The fault content is obtained by multiplying the number of developed executable lines of code by the fault density. The relationships are given by:

$$\beta = B\frac{\lambda_0}{\omega_0} \qquad \textbf{(7.2)}$$

$$\nu_0 = \frac{\omega_0}{B} \qquad \textbf{(7.3)}$$

where B is the *fault reduction factor*. This parameter is sometimes called the defect removal efficiency.

The fault reduction factor parameter should be estimated based on collected project data whenever possible. Suggested defect removal efficiencies for Levels of the CMM are indicated in Table 7-11[6]:

TABLE 7-11. Suggested Defect Removal Efficiencies for SEI CMM Levels

SEI CMM Levels	Removal Efficiency
SEI CMM 1	0.85
SEI CMM 2	0.89
SEI CMM 3	0.91
SEI CMM 4	0.93
SEI CMM 5	0.95

Performing a software reliability prediction, time-adjusted by the growth model, provides a continuous customer-oriented assessment of software quality the end-user can expect to experience if the software is released at a given future date. Reliability planning and management are facilitated by use of the software reliability growth model, which can be interpreted in different ways to derive various quantities of interest. A few of the more important ones are described here.

The most important parameter is the failure rate (failures/CPU hr.). A software reliability growth model (see 8.4) describes the decline in the software failure rate that occurs during the system growth

[6] "Measuring Global Software Quality", Software Productivity Research, Capers Jones, Burlington, MA, 1995.

phase as the number of faults in the code declines. Let $\lambda(\tau)$ be the instantaneous failure rate at time t. The failure rate at the start of system test is denoted $\lambda_0 \equiv \lambda(0)$.

Management may be interested in calendar time parameters. The relationship between calendar time, denoted t, and execution time, denoted τ, during system test, is governed by a resource-limiting parameter: failure identification personnel (testers), failure resolution personnel (debuggers), or computer time. The calendar time component allows a schedule to be established relating the amount of time (in weeks) needed to reach a failure intensity objective. The method for mapping execution time to calendar time is detailed in Section 8.

The software reliability growth model, once its parameters are determined, provides one-to-one mappings between any two of the following quantities: execution time, calendar time, failure rate, and expected cumulative number of failures.

7.4 Forecasting Failure Rate Versus Execution Time.

A reliability growth model can be used to forecast the failure rate the software will exhibit at any time t into system test. A well-known reliability model is the Musa execution time reliability growth formula:

$$\lambda(\tau) = \beta \nu_0 \exp[-\beta\tau] \quad \textbf{(7.4)}$$

The function

$$\ln \lambda(\tau) = \ln \beta \nu_0 - \beta\tau \quad \textbf{(7.5)}$$

will plot as a straight line on semi-log paper. If the software code is frozen and is operational at time t, the software may then exhibit a constant failure rate λ. The reliability function is then $R(\tau') = \exp[-\lambda\tau']$, where τ' is execution time measured from the present.

7.5 Forecasting Cumulative Failures Versus Execution Time.

A method for predicting the expected cumulative number of software failures that will be experienced in system test through time t is given by

$$\mu(\tau) = \nu_0 (1 - \exp[-\beta\tau]) \quad \textbf{(7.6)}$$

See Section 7.3.4 for a discussion of the input parameters to this model.

7.6 Forecasting When a Reliability Objective Will be Met.

The growth model can answer many useful planning questions. For example, when will a failure rate requirement or some intermediate failure rate objective λ_F be met? From the Musa reliability growth model, the number of failures that must be experienced to reach that objective is

$$\mu = \nu_0 \left(1 - \frac{\lambda_F}{\lambda_0}\right) \quad \textbf{(7.7)}$$

where ν_0 is one of the two parameters of the software reliability growth model. The amount of execution time to meet that failure rate objective is

$$\tau = \frac{1}{\beta} \ln \frac{\lambda_0}{\lambda_F} \quad \textbf{(7.8)}$$

where β is the other growth model parameter.

<u>Steps.</u>

A. Predict or estimate the software reliability growth model parameters β and ν_0.

B. Predict or estimate the initial failure rate λ_0. Note that

$$\lambda_0 = \beta \nu_0 \quad \textbf{(7.9)}$$

C. Determine a failure rate objective λ_F.

D. Determine the expected number of software failures that must be experienced from equation **(7.7).**

E. Determine the amount of execution time to reach λ_F using equation **(7.8).**

<u>Example:</u>

Suppose that the initial failure rate has been predicted to be λ_0 = 18 failures per CPU hour. The software reliability growth model parameters have been predicted at ν_0 = 139, therefore

$$\beta = \frac{\lambda_0}{\nu_0} = \frac{18}{139}$$

The expected number of failures that must be experienced to reach λ_F = 1 failure per CPU hours is

$$\mu = \nu_0 \left(1 - \frac{\lambda_F}{\lambda_0} \right) = 139 \left(1 - \frac{1}{18} \right) \approx 131$$

The execution time required is

$$\tau = \frac{1}{18/139} \ln \frac{1}{18} \approx 22.317 \text{ CPU hours}$$

7.7 Additional Failures and Execution Time to Reach a Reliability Objective.

The growth model can also estimate the incremental number of failures or amount of execution time to get from a present failure rate λ_P to a failure rate objective λ_F. The additional number of failures that must be experienced to go from λ_P to λ_F is

$$\Delta\mu = \frac{1}{\beta}(\lambda_P - \lambda_F) \qquad \textbf{(7.10)}$$

See Section 7.3.4 for a discussion of the input parameters to this model. The additional execution time $\Delta\tau$ that is required to reach the failure rate objective is

$$\Delta\tau = \frac{1}{\beta}\ln\frac{\lambda_P}{\lambda_F} \qquad \textbf{(7.11)}$$

Steps.

A. Start with a present failure rate λ_P and a failure rate objective λ_F.

B. Determine the software reliability growth model parameter β.

C. Obtain the additional number of failures that must be experienced to go from failure rate λ_P down to failure rate λ_F, use formula **(7.10).**

D. To determine the additional amount of execution time to reach λ_F, use formula **(7.11).**

Example:

Suppose that the present failure rate is λ_P=22 failures per hour and an intermediate failure rate objective is λ_F=8 failures per hour. The software reliability growth model parameter β has been estimated at 0.6.

A. Obtain the additional number of failures that must be experienced to get from the present failure rate λ_P to the future failure rate objective λ_F:

$$\Delta\mu = \frac{1}{\beta}(\lambda_P - \lambda_F) = \frac{1}{0.6}(22 - 8) \approx 23$$

B. Obtain the additional execution time to get from λ_P to λ_F through the formula

$$\Delta\tau = \frac{1}{\beta}\ln\frac{\lambda_P}{\lambda_F} = \frac{1}{0.6}\ln\frac{22}{8} \approx 1.686 \text{ hours.}$$

Project management generally thinks in terms of calendar time (t) rather than execution time (τ). Determining the relationship between the two is described in Section 8. The quantities derived from the prediction and growth models feed back into the software development process to provide systematic planning for and control of reliability achievement as a function of calendar time.

Project management can set intermediate reliability goals based on the predicted reliability figure and growth rate. If later predictions show that an intermediate goal is not likely to be met, management can re-allocate resources based on comparison between the planned and assessed reliability figures.

An important schedule determinant once the software is in system test is the ratio between execution time and calendar time. This ratio is determined by the limiting resource. The resources involved in system test are failure identification personnel, who perform the testing; failure resolution personnel, who debug the programs; and computer time.

If too many failures are experienced, the failure resolution personnel will become backlogged with debug work, holding up testing. If setting up test cases and analyzing the output for failures takes a long time, then the failure identification personnel are the limiting resource. If one category of personnel is the limiting resource, then overtime may be an appropriate solution. It is important to remember that the management decisions are based on the achievable reliability.

7.8 Optimum Release Time.

There are methods available for predicting the optimal release time. Table 7-12 summarizes these methods:

TABLE 7-12. Methods For Predicting Optimal Release Time

Method	**Description**
Musa model	Based on software reliability growth.
Process Productivity Parameter[7]	Developed by Quantitative Software Management, Inc. Can predict optimal release time based on current productivity, effort and size of product.
COCOMO model[8]	Developed by Barry Boehm. Based on size, schedule time and effort as well as some product and development characteristics.

The Musa software reliability growth model discussed previously can be used to determine the optimum release time for minimizing overall cost. Each failure during development entails a cost c_1.

Each failure in operational use entails a cost c_2 (the failure costs can be broken out by failure severity category). Additionally, there is a cost c_3 for each time unit of system test. The total cost of system test can be computed as follows:

If the software is hypothetically released at time t_e, the cost attributed to system test failures is

[7] "Measures for Excellence", Larry Putnam, Ware Myers, Quantitative Software Management, Yourdon Press, Englewood Cliffs, NJ, 1992.

[8] "Software Engineering Economics", Boehm, Barry, Prentice Hall, Englewood Cliffs, NJ, 1981.

$$D_1(\tau_e) = c_1 \bullet \mu(\tau_e) \quad \textbf{(7.12)}$$

The cost incurred by failures during operation is

$$D_2(\tau_e) = y \bullet \lambda(\tau_e) \bullet c_2 \quad \textbf{(7.13)}$$

where y is the number of time units of operation. The cost incurred by system test is

$$D_3(\tau_e) = \tau_e \bullet c_3 \quad \textbf{(7.14)}$$

The total cost when the software is released at time t_e is the sum of the three costs:

$$D(\tau_e) = D_1(\tau_e) + D_2(\tau_e) + D_3(\tau_e) \quad \textbf{(7.15)}$$

The optimum time to release the software, from a pure cost point of view, is found by minimizing the function $D(t_e)$.

$$(\tau_e)_{\min} = \frac{\ln \dfrac{y c_2 \beta^2 \nu_0 - c_1 \beta \nu_0}{c_3}}{\beta} \quad \textbf{(7.16)}$$

<u>Steps.</u>

A. Based on labor, overhead, and related expenses, determine the cost per failure c_1 for failures that occur during system test, as well as the cost per unit of execution time, c_3.

B. Based on program maintenance, service impact, and related expenses, determine the cost per failure c_2 for failures that occur during operational use. Determine the operational life of the system, y.

C. From prediction or growth testing, determine the software growth model parameters β and ν_0.

D. Compute the minimum-cost release time using equation **(7.16)**

Example:

Suppose that the cost of a failure during system test is c_1 = $1000, and that the cost of a failure during operation is $6000. Each day of system test costs $4500. The growth model parameters are

$$\beta = 0.002, \ \nu_0 = 120$$

Using the day as the unit of time, the minimum-cost release point is

$$\tau_e = \frac{\ln \dfrac{y\, c_2\, \beta^2\, \nu_0 - c_1\, \beta\, \nu_0}{c_3}}{\beta}$$

$$= \frac{\ln \dfrac{(1825)(6000)(0.002\ )^2\, (120) - (1000)(0.002)(120)}{4500}}{0.002}$$

$$= 54 \ \text{days}$$

7.9 Ultra High Reliability Prediction.

It is essential to consider achievability and testability when predicting reliability for software systems that must be relatively high. Demands for perfection should be avoided as they are not testable or demonstrable. For example, if the demand for the failure rate is 10^{-4} then there must be sufficient resources for extensive validation and verification to demonstrate this level. The current state of the art is limited in providing any help in assessing the software reliability at this level.

Techniques such as *Formal Methods*[9] are currently being used by software organizations developing ultra high reliability systems. Refer to the Appendix of this notebook for more information on ultra high reliability prediction.

[9] Refer to “Software Engineering A European Perspective”, Richard Thayer and Andrew McGettrick, IEEE Computer Society Press, Los Alamitocs, CA, 1992, as well as “The Cleanroom Approach to Quality Software Development” by Michael Dyer, John Wiley & Sons, Inc, NY, 1992.