# Empirical and Face Validity of Software Maintenance Defect Models Used at the Jet Propulsion Laboratory

William Taber
California Institute of Technology
Jet Propulsion Laboratory
Mission Design and Navigation Software Group
William.L.Taber@jpl.nasa.gov

Dan Port
University of Hawaii
Shidler College of Business
Information Technology Management
Telephone number, incl. country code
dport@hawaii.edu

## ABSTRACT

**Context:** At the Mission Design and Navigation Software Group at the Jet Propulsion Laboratory we make use of finite exponential based defect models to aid in maintenance planning and management for our widely used critical systems. However a number of pragmatic issues arise when applying defect models for a post-release system in continuous use. These include: how to utilize information from problem reports rather than testing to drive defect discovery and removal effort, practical model calibration, and alignment of model assumptions with our environment.
**Goal:** To show how we can develop confidence in the practical applicability of our models for obtaining stable maintenance funding.
**Method:** We describe the strong empirical and face validity we have investigated for our maintenance defect discovery and introduction models. We discuss the practical details of calibration and application within a functioning maintenance environment.
**Results:** We find that our models, despite their simplicity, appear quite valid.
**Conclusions**: The models are useful in justifying and obtaining stable maintenance funding.

## Categories and Subject Descriptors
D.2.4 Software/Program Verification: Reliability

## General Terms
Management, Measurement, Performance, Design, Economics, Reliability, Experimentation.

## Keywords
software defect model, software maintenance, software reliability

## 1. INTRODUCTION

The software used for NASA's deep space missions is critical to the success of those missions. For many of these missions, defects encountered and not repaired in a timely manner not only inhibit operations, but may also lead to the loss of a billion dollar mission. Despite our many years of experience focusing on quality development and control, achieving zero defects in a delivered system has evaded us. We accept that defects in software are not just a possibility – they are a certainty. As a result, for every system in operation we must plan for maintenance effort to find and repair these inevitable defects. Given our limited budgets and staffing resources, and often a time critical need for repair, maintenance effort cannot be planned ad hoc. It is simply too risky to have a critical system non-operational for an arbitrary or unknown period of time. Many tough experiences have shown us that poor planning for maintenance increases risk of mission failure.

Good maintenance planning must address questions of how many defects we may encounter, the likelihood of their surfacing at a critical moment, and how long it will take to discover and correct them. These questions require having *credible* and *pragmatic* maintenance defect models. While there are many potentially useful models suggested in the literature, none are widely accepted or standard for use in industry [6]. This in part may be due to pragmatic demands rather than theoretic concerns such as:

- Do the models provide useable and useful information for maintenance planning?
- Are the models practical to calibrate and use on our projects?
- Are the assumptions of the models reasonably valid and validatable relative to our environment?

The general question is, "Can we have reasonable confidence in *using* the models for maintenance planning for our systems?" So while there is a great deal of theoretic study on the derivation and effectiveness of defect models, this work aims to explore the pragmatic concerns indicated above. For example our model needs differ as described in [7] - the systems are widely-used, and under multi-release post-development where we must make use of user-reported defects (i.e. problem reports) rather than test data. For this we have selected based on principles suggested in the maintenance literature ([2][5][6]) models that follow *exponential decay* (i.e. finite exponential). We explore the face validity of applying these models to two kinds of critical systems in maintenance at NASAs Jet Propulsion Laboratory (JPL). We discuss how these models provide a useful and practical mechanism for understanding the accumulation and removal of defects in final-release systems as well as how defect accumulation and removal evolves for systems with planned continuous releases. We show empirically for our two kinds of systems that these models are practical to calibrate and fit remarkably well. Moreover, the models are not merely empirically good fits, but have high face validity and are based on assumptions about how defects are distributed and are discovered that are in good alignment with our intuition and experience with maintenance defects occurrence and removal.

Finally we discuss how the JPL Mission Design and Navigation Software Group uses these models to determine our defect rate and to inform our users about the reliability of a release and how that reliability will improve through a cooperative process of testing, reporting problems and releasing repairs quickly. In addition, we use these models to ensure that senior managers are informed both on principle and empirically about the inherent risks in software maintenance and that schedules and budgets must accommodate the defect decay process to reach a desired level of reliability in operation of these systems.

## 2. DEFECT ACCUMULATION MODEL: POST-FINAL RELEASE

*Software maintenance* is the ongoing activity performed on a system post-delivery and ostensibly in operation. It differs from pre-delivery development in a number of ways in so far as how defects are discovered and removed. One fundamental difference is in the stability of requirements and a deliberate resistance to introducing continuous changes to the system. There are two primary maintenance profiles – (A) static requirements where new development is neither expected nor planned, and (B) continuous improvement and releases where new development is planned and expected according to managed releases. We begin by considering the assumptions we might make for modeling defects under maintenance profile (A):

*Assumption 1: The rate of change in the expected number of defects in the system, N(t), is proportional to the number of defects in the system.*

*Assumption 2: The software is changed only by repairing defects as they are discovered.*

*Assumption 3: The software is executed approximately the same number of times each day by a fixed population of users.*

*Assumption 4: Defects are uncorrelated and are distributed randomly through the code at a rate proportional to the code size.*

If we let $T$ be some reference time, then the assumptions above leads to a simple defect decay model for $N(t)$:

$$N(t) = N(T)e^{-\alpha(t-T)}. \quad (1)$$

The parameter $\alpha$ represents the proportionality suggested in Assumption 1 and is commonly referred to as the *defect decay constant.*

Although simple in form, the number of expected defects at time $T$ is *not observable* from the history of defects reported and repaired in a software system. In particular, we cannot know the number of defects present in the system at delivery (i.e. when maintenance begins). This makes the simple model suggested in (1) unusable in practice. However we can observe the number of defects that have *accumulated* through defect reports (or user generated problem failure reports) at any moment in time. This is the number of defects at time $T$ minus the number of defects remaining at time $t$ or $A(t)=N(T)-N(t)$. That is,

$$A(t) = N(T)(1-e^{-\alpha(t-T)}). \quad (2)$$

This is a finite exponential model that has been suggested frequently in the defect modeling literature [2][4][5][6][7]. The defect accumulation model has the useful property that it is not sensitive to the reference time $T$. Past knowledge of defects accumulation is not required to model the maintenance defect behavior of a system. It does not matter whether a software maintenance effort has a good or poor record of the discovery of defects in the past. The project can start accurately recording defect discoveries at any point and use the model to estimate the defect decay rate and number of defects remaining in the system. Hence this model is practical to use and provides useable information.

### 2.1 Estimating $\alpha$ and $N(T)$

To make use of the decay and accumulation models we must be able to obtain an estimate of the defect decay constant $\alpha$ and for the initial number of defects $N(T)$ when we begin careful observation of the defect history. Details on practical means for obtaining such estimates are not specified in the literature so we will discuss this here. For the sake of simplicity in computation, we set the initial reference time $T=0$, and our initial number of remaining defects at this time $N=N(0)$. Suppose that we have observed $n$ distinct defects at times $\{t_1, t_2, \ldots, t_n\}$ where the $k$'th defect is observed at time $t_k$. According to our model (2) the expected number of defects accumulated at time $A(t_k) = N(1-e^{-\alpha t_k})$. We can view defect discovery as a statistical control process where over time we expect that the variation of the actual count from the predicted count $k$ will have Normal variation associated with sampling. If this assumption is correct we should be able to estimate both $N$ and $\alpha$ by minimizing the variation in the squares between the number of predicted defect observations and the actual number of observations. That is, we can apply the method of least squares and find those values of $N$ and $\alpha$ that minimizes the sum

$$\sum_{k=1}^{n} \left(k - N(1-e^{-\alpha t_k})\right)^2. \quad (3)$$

One approach we might take is to apply ordinary least squares regression, possibly transforming the data so that (4) is a linear. The complication with this is that $N$ is an integer and this is difficult to constrain using ordinary regression methods. We note that maximum likelihood estimation is potentially another approach to estimating the parameters but this too suffers this same issues and has a less straightforward to interpret optimization criteria. Fortunately it's not too difficult to use integer programming to estimate $N$.

We start by finding an initial estimate for $\alpha$ by looking at what the accumulation model predicts for the time of the discovery of the $k$'th defect. Using (2) we can solve for this time $\tau_k$ (i.e when the $k$'th defect is expected to be observed):

$$k = N\left(1-e^{-\alpha\tau_k}\right)$$
$$\tau_k = \frac{1}{\alpha}\ln\left(\frac{N}{N-k}\right) \quad (4)$$

We treat the values $\tau_k$ as the predicted times at which the defect accumulation is incremented. We can then vary $N$ and $\alpha$ to minimize the square of the deviations between predicted $\tau_k$ and observed times when defect accumulation is incremented $t_k$. That is, minimize

$$\sum_{k=1}^{n}\left(t_k - \frac{1}{\alpha}\ln\left(\frac{N}{N-k}\right)\right)^2 . \tag{5}$$

For a given $N$ this expression is minimum when

$$\alpha = \frac{\sum_{k=1}^{n}\left[\ln\left(\frac{N}{N-k}\right)\right]^2}{\sum_{k=1}^{n} t_k \ln\left(\frac{N}{N-k}\right)} . \tag{6}$$

Since $N$ is the number of defects in the system at time 0 and an integer, we know that $N \geq n$. It is a simple programming exercise to iterate over values of $N$ (starting with $N = n$), computing and evaluating (5) until an optimal value of $N$ and associated value of $\alpha$ is obtained. This value of $\alpha$ is good enough to estimate the optimal $N$ but likely not optimal with respect to (3). Having obtained $N$ we can apply the transformation $Y_k=\ln(1-A(t_k)/N)$ to linearize (2) and apply simple linear regression forced though the origin on $(t_i, Y_i)$ to determine an optimal value for $\alpha$ in the defect decay model.

From the discussion above we see that estimating the parameters for (2) from observed defect accumulations is both straightforward and practical. This will be illustrated with two case studies in the next section.

## 2.2 Empirical Validation: JPL Legacy System

The model developed above has been compared against defect data accumulated by the Mission Design and Navigation Software Group at the Jet Propulsion Laboratory in Pasadena, California. This system is composed of over one million logical lines of heritage FORTRAN code. The initial development of this system began in the late 1960s and continued until approximately 2006. During the period from 1999 until the present, an effort was undertaken to replace the legacy navigation system with a new system, Monte, to be developed in modern languages using modern software development methodologies. As a result, in early 2006, the legacy system became a static (final release) system. Defects in the system have been repaired, but aside from this no new code has been introduced into the system. In addition, the software has been in regular daily use by a nearly constant population of users during this time. The legacy navigation software system is a large system that fits the assumptions for the defect decay and accumulation models as described previously.

Beginning in 2002, the software group has used the open source defect tracking tool *bugzilla* to capture all anomalies in the operation of navigation software. One of the important features of *bugzilla* from the point of view of modeling defect discovery is that all defect reports as well as updates to those reports are time-stamped using the host system's clock. Moreover, all users of the software are trained in the use of *bugzilla* and use it to report any suspected problems in the software. This provides reliable time of defect occurrence data as needed for calibrating our models.

Given that the model (2) is derived from basic assumptions that appear plausible for our system, it in principle has high face validity. That is, we have some rational basis for selecting this model. While this provides a degree of comfort, the concern here is about how reasonable or valid these assumptions are in practice for our system. We will look at some empirical data and our experience on our maintenance projects to explore the validity of these assumptions.

Assumption 2 is straightforward. For our legacy system, by mandate, the system is only changed in response to problems reported. New requirements or requests for additional capabilities are prohibited.

Assumption 3 is a little more difficult to justify. We note that the adoption of Monte by the flight projects was a slow process and required continuous management pressure. Flight projects are reluctant to accept new software. They readily accept bug fixes and small additions, but entirely new systems are a major challenge. During the period in question the navigation staff was very stable and nearly all were employed full time on flight projects: Cassini, Dawn, Deep Impact, Genesis, Hayabusa, MER, MRO, Odyssey, Spitzer, Stardust. The user base for Monte was until relatively recently a different population of users. These were the early adopters and for the most part the new people who "grew up" with Monte and never learned the legacy tools. Monte has now successfully replaced the legacy product on all flight projects but the transition was not complete until Cassini adopted it in 2012. As a result of these observations, the assumption that the population of user is fixed is reasonable. During that period, the legacy software suite of tools (over 300 applications) was exercised an average of 10,000 times/day for study purposes by the legacy users.

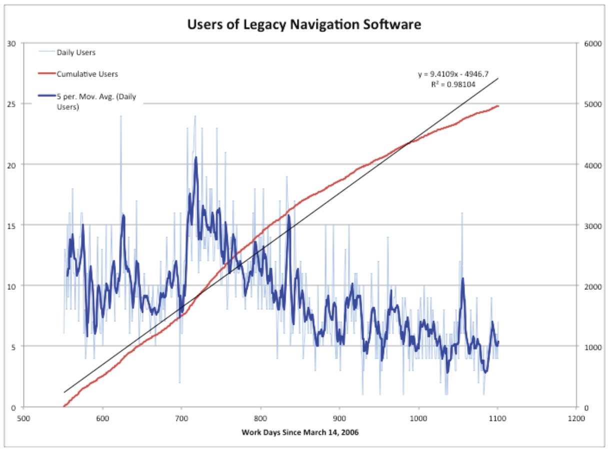

**Figure 1a. Per Day Usage of Legacy Navigation System.**

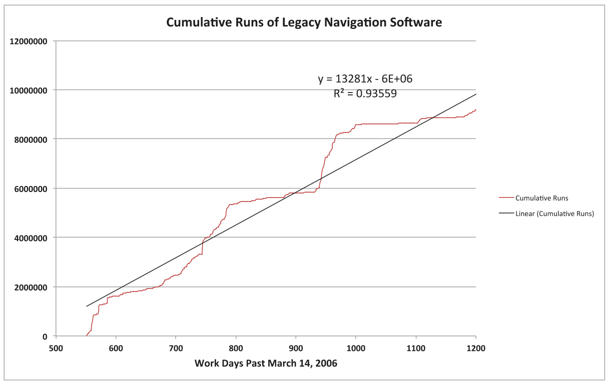

**Figure 1b. Runs of Legacy Navigation System.**

We note that that the system use rate is not exactly constant over the period in question and there are spikes when people are doing parallel jobs. The main issue in satisfying Assumption 3 is a uniformity of usage and low variance, which is supported by the usage data in Figure 1a showing the per day number of users and cumulative users of the legacy system. In terms of relatively constant system use, for non-operations work, we have accounting logs since 2008 on our computing cluster. Figure 1b shows the that from these logs the cumulative runs of the legacy system is approximately linear (with obvious "seasonality") indicating that the system general has a more or less constant use rate.

For assumption 1, consider the operational bugs graph in Figure 2. Visually is it apparent that the rate of defect discovery is proportional to the number of operational bugs. Comparing this graph with the code size growth we see that defect rate is also proportional to code size. The linear increase in operational bug discovery over time is consistent (but does not prove) with randomly distributed, uncorrelated defects. This also is consistent with the human nature of software development. To contrast this, we observe in Figure 2 a non-linear growth in bugs discovered in developing Monte (the replacement for the legacy system), which are generally regarded as non-random and highly correlated.

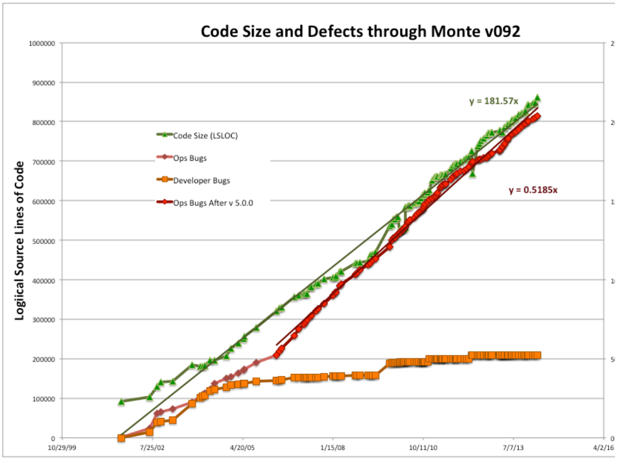

**Figure 2. Defect Rates and Code Size in Monte.**

To test the defect decay/accumulation model, the creation time-tags of the last 101 defect reports were obtained from the *bugzilla* defect history database. Using the defect from the first of these reports as a reference time, a set of time offsets for defect discovery has been constructed. These offsets are from a normalized timeline that removes all non-work hours—evenings, holidays and weekends are all removed from the time-line. A Microsoft Excel Workbook has been developed for computing the best model fit to the *bugzilla* defect report history. The spreadsheet follows the parameter estimation method outlined in Section 2.1 earlier. Figures 3 and 4 below show the results of this fitting process for two different systems in post-delivery maintenance.

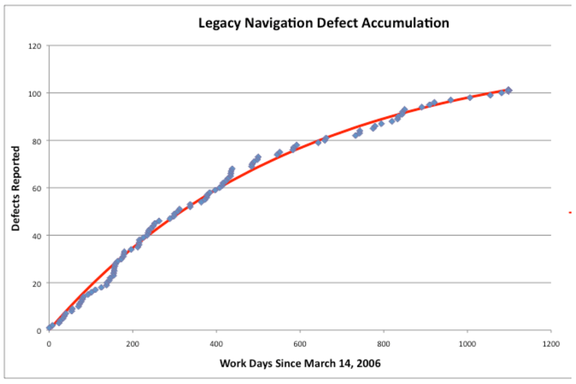

**Figure 3. Fit of the accumulated defect history for legacy navigation software.**

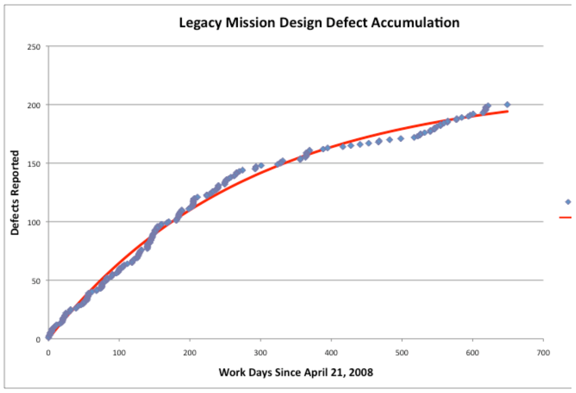

**Figure 4. Fit of the accumulated defect history for Mission Design software.**

In addition to constructing an accumulation model, a decay model is constructed for the observed defect history. Plots for these models are shown in Figures 5 and 6 below.

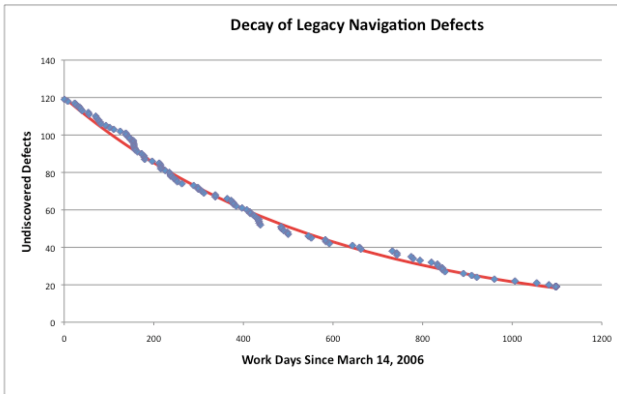

**Figure 5. Model for the decay of defects in Legacy Navigation software.**

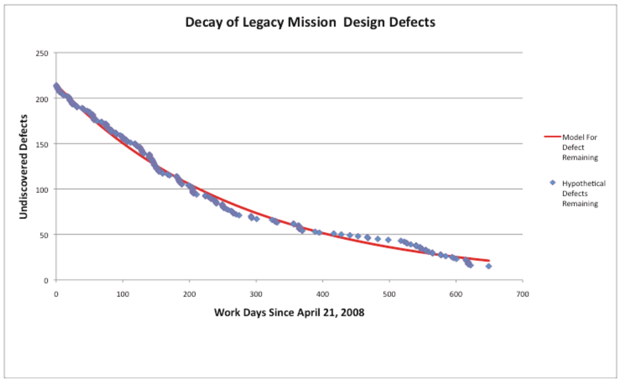

**Figure 6. Model for the decay of defects in Legacy Mission Design software.**

As is apparent from the above figures, the models fit the observed defect discovery history remarkably well. The R-square values are all over 0.95 and while other models may indeed also fit quite well, we note that our models are the simplest of these in terms of form and number of variables that match the assumptions indicated. Again form a pragmatic perspective it is difficult to justify utilizing more complex models when the current models appear to be sufficient.

DEFECT ACCUMULATION – CONTINUOUS RELEASE

Most software systems are not final releases and hence are under development during maintenance. In addition to staged capability development, user needs evolve and the systems are augmented with new features to meet those needs. This is particularly true in iterative release life cycle strategies. Here an initial version of the software is released to the user community that meets enough of user needs to make the system usable. Development continues with releases being made as soon as the development team feels new features are ready for use or in planned stages. Hence we must relax Assumption 2 and understand its defect model implications. By considering the limiting state we can model such a system as having continuous rather than discreet releases of the software.

If the software is released to a fixed user population that uses the software at a uniform rate, and defects are repaired as they are discovered, we can model the rate of change in the number of defects in the system as the rate of change we would see if development stopped *plus* the rate at which defects are added to the system. In addition if the development team has a fixed size, the rate of code production is nearly constant. We can also assume that over time the number of defects introduced per new line of code is nearly constant as well. Thus for a fixed development team working at a constant rate, the expected number of defects produced per unit time $D$ is a constant. Under these conditions our maintenance defect decay model becomes:

$$N(t) = \frac{D}{\alpha} + \left(\frac{\alpha N(T) - D}{\alpha}\right) e^{-\alpha(t-T)} \tag{7}$$

As before, we can't observe $N(t)$, but we can observe the accumulation of defects $A(t)$:

$$A(t) = D(t-T) + \left(\frac{\alpha N(T) - D}{\alpha}\right)\left(1 - e^{-\alpha(t-T)}\right) \tag{8}$$

We now consider some empirical validation of these models.

## 2.3 Empirical Validation: Next Generation Navigation System (MONTE)

Given that our system is not literally under continuous release, and that we could not collect continuous release data anyway, we must look for defect behaviors our system exhibits and see if they are what are predicted by our models. Models (7) and (8) imply the following defect behaviors for a system under continuous release and the system is used at a uniform rate:

A. The number of defects in the system will stabilize to the rate of production of defects $D$ divided by the discovery rate $\alpha$.

B. The defect accumulation will become linear over time with a rate of discovery equal to the rate of production.

The first behavior (A) is not directly observable. We never really know the number of defects in the system. However, since we can observe the accumulation of defects from a system under continuous development, we can see if the accumulation appears to be linear and perform a reasonableness test on the rate of accumulation to see if it gives us a plausible estimate for the rate of defect production.

As was discussed earlier, the development of software to replace the legacy system used for mission navigation was begun in 1999. This new system has had frequent releases since it began so that developers could ascertain usability and gather feedback from users on needed features. The system was first placed into a mission environment in May of 2006. By this date, a stable base of users had been established. Moreover, since the software was used in a mission environment; its use was similar to the usage of the legacy navigation system. At the same time, new features continued to be added to the system to support the wide range of needs for the various trajectory design and navigation demands of NASA's deep space missions.

Since May of 2006, there have been over 70 software releases of the software (on average a new release about every 4 weeks) and the system has experienced uniform usage by the user community. Hence, the development and usage of the system since 2006 is approximately a continuous release system.

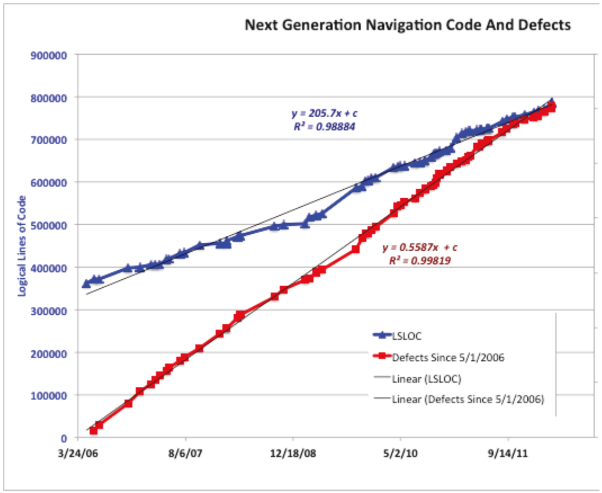

**Figure 7. Code size and defect discovery history for next generation mission design and navigation software.**

Figure 7 shows both the accumulation of deliverable lines of code in the system as well as the accumulation of defects for the released system. As can be seen, the rate of accumulation of defects is highly linear. Moreover, the ratio of the defect slope to the code size slope yields a defect density of 2.7 defects per thousand logical lines of code—a very plausible defect density for this project.

## 3. A PRIORI DEFECT DISTRIBUTION FOR THE NUMBER OF DEFECTS IN A SOFTWARE RELEASE

As observed earlier, the number of defects in software can be approximated as a linear function of the number lines of code—if the size of the source base doubles the number of defects approximately doubles. If defects are tracked consistently over time, we can soon learn the number of defects typically produced for a given size software application. This rate is usually measured in defects per thousand lines of code (KSLOC). If the software development environment is stable, and the developing organization has a reasonably long track record of development, we can multiply the size of a release by the rate of defect production to obtain the expected value for the number of defects in the code.

From long work with the navigation software, our intuition is that most defects found in maintenance are unrelated. This is significantly different than defects found during development so to check whether or not our intuition is reasonable, we examined our change records for the software in response to defect reports. Whenever a defect is repaired, the submission of new files to the configuration management system is tagged with the identifier for that defect. As a result, it is possible to measure the overlap between defects statistically. We do this in the following manner. Two defects are regarded as being related if their repair affects a common source file within some prescribed specified time interval. Since repaired software is made available to users within a week of being repaired, we picked a time threshold of 30 days. Of course, it is quite possible that unrelated defects will touch the same source file within 30 days, so we may very well generate some false positives by this technique. However, even with the possibility of these fall positives, we find that most defects are indeed isolated. When we use this method of characterizing defects as non-isolated, we find that for all software (both the legacy and new software), fewer than 12% of defects are not isolated.

Hence it seems reasonable to assume that defects in one part of the code are typically independent from defects in other parts of the code. We can use this observation together with the linear relationship between code size and defects to select an *a priori* distribution for the number of defects in a software release.

Suppose that we know the defect density that a given development achieves, and that for some release of the software we have computed the number of expected defects from this density to be *β*. Under the basic assumptions discussed previously we can stipulate selecting a Poisson density function to represent the likelihood of having *n* defects:

$$\Pr(n \text{ defects}) = \frac{\beta^n}{n!} e^{-\beta} = P_\beta(n) \qquad (9)$$

Given the exponential defect discovery model (1) and the assumption of independence of defect discovery, we see that the probability of discovering a defect after time *T* given that *n* defects have already been observed is:

$$\Pr(t > T \mid n) = e^{-n\lambda T} \qquad (10)$$

We can combine (9) and (10) by applying Bayesian inversion on (10) giving:

$$\Pr(n \mid t > T) = \frac{\left(e^{-\lambda T}\beta\right)^n}{n!} e^{-\left(e^{-\lambda T}\beta\right)} = P_{e^{-\lambda T}\beta}(n) \qquad (11)$$

The above is the a-priori distribution of observing *n* defects after a release at time *T*.

### 3.1 Empirical Validation: Can We Predict Maintenance Need?

Here again since we can never know how many defects are present in the system we cannot collect data to directly empirically validate models (9) and (11). So while the theory behind the models presented here is relatively simple and lends itself to simple calculations, the question remains "Are the models good enough to aid in guiding the management of a software development effort?" We again look to what defect behaviors these models would predict as our means to validate them. Using the Poisson model (9) for the initial a priori distribution of defects in a new software release and the decay model for their discovery, we should be able to fit the observed history of defect discoveries. We do this for the next generation navigation software system, Monte.

Recall that under the exponential decay model that the total number of undiscovered defects in the system under a continuous release model will stabilized to the value *B* given by $B = D/\lambda$ where *D* is the defect production rate and $\lambda$ is the decay constant. Under the Poisson distribution model for the number of defects in a release, we can predict what the distribution is for undiscovered defects. Given the defect density, $\rho$ for the system we expect the number of distributions for the steady state number of defects to be Poisson with mean $S\rho - A$ where *S* is the size of the software and *A* is the number of defects that have been discovered. For the system in question we approximate $\rho$ from the ratio of the slope of code production divided by slope of defect discovery. This yields a value of 2.692 defects per 1000 logical lines of code.

To date 2036 defects have been discovered in the code base of 860,889 logical lines of code. Thus the model distribution for the number of defects remaining in the system is Poisson with mean equal to $B = 861 \cdot 2.7 - 2036 = 288$. The historical discovery rate for defects is 0.742 defects/workday. From these values we can produce an initial estimate for $\lambda = D/B = 0.742/288 = 0.00257/\text{workday}$.

With an initial guesses in hand for $\lambda$ and $D$ we can simulate the history of defect discovery for the new system and adjust these initial values to find a best fit between theory and the observed accumulation. When this this is done, we find the "best" values for our theory parameters are: $\lambda = 0.002494$, $D = 2.868$. The simulated history and theoretical history are shown in Figure 8 below.

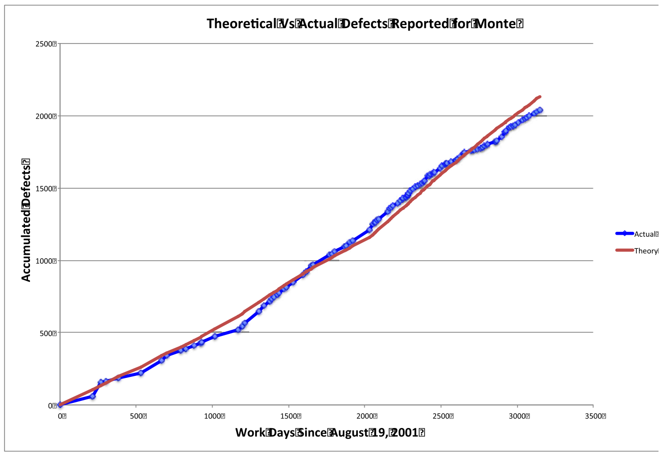

**Figure 8. A-priori versus actual accumulated defects in next generation navigation software.**

Moreover, if one creates an empirical distribution for the set of possible histories via monte carlo simulation we find that the actual history falls well within the densest set of possible histories as seen in Figure 9:

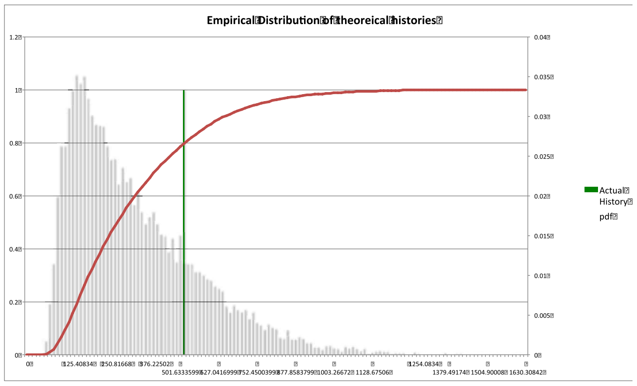

**Figure 9. Distribution of simulated defect histories.**

The a-priori defect distribution appears to predict well defect accumulation and is consistent with the kind of defect density histories we expect to find based on the project parameters.

## 4. CONCLUSION

The models discussed here have been used in the Navigation Software Group at the Jet Propulsion Laboratory to provide insight into the reliability of our software. Flight projects are risk averse when adopting new software. Their first question is always "Does it meet our requirements?" The second question is "How buggy is it and when will we be confident that we've found and fixed the important ones?" Prior to the development of these models we could only point to our track record of past deliveries. However, with these models we have been able to provide flight projects with a more quantitative assessment of the risk of accepting new software. Using the decay model together with the Poisson distribution to characterize the new defects released in software, the software development team has been able to work with flight projects to identify the level of resources they need to devote to maintaining reliable operation the new software.

In addition to providing flight projects with better confidence in reliability, these models make it much easier to justify the budget and staffing required to maintain reliable operations of a system. Prior to the development of these models, the budget for sustaining a maintenance effort was under constant threat. There was a natural tendency for the software development team to request more than was needed. And there was a natural tendency for the sponsoring programs to provide too little. With the advent of these models, both the program and the development groups have been able to reach a comfortable agreement on the resources needed to sustain the institutional investment in our mission critical navigation and trajectory design software. This does not mean that budgetary pressures have been removed, but both sides of the budget issue can now address the issue from the perspective of a quantified level of risk and debate can focus more on acceptable risk tolerance rather than speculation on reliability.

The models presented are selected on principles and backed by a small sample of defect accumulation histories. Other models may also fit our observations well. However, the models presented here are simple requiring only two parameters that are straightforward to estimate: a defect density and a rate of decay in the discovery of defects. The defect observations from the projects we have presented are consistent with these models to within a few percent so that deviations from the model are easily within the noise one would expect from a stochastic process.

## 5. ACKNOWLEDGMENTS

This research was carried out at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration.

## 6. REFERENCES

[1] Casella G, Berger R. *Statistical Inference 2nd Edition*. Duxbury Press. 511 Forest Lodge Road, Pacific Grove, CA 93950, USA

[2] Garrison Q. Kenny. "Estimating defects in commercial software during operational use." IEEE 1993.

[3] J.D. Musa. "Validity of Execution-Time Theory of Software Reliability." IEEE Transactions of Reliability, Vol R-28, 1979. pp181-191. J.D. Musa, K. Okumoto

[4] Jelinski, Z., and Moranda, P. "Software Reliability Research," *Statistical Computer Performance Evaluation,* Freiberger, W. Ed. Academic Press, New York, NY

[5] Kan S. *Metrics and Models in Software Quality Engineering 2nd Edition.* Addison-Wesley, Boston MA, USA

[6] Li, P., Mary Shaw, and Jim Herbsleb. "Selecting a defect prediction model for maintenance resource planning and software insurance." *EDSER-5 affiliated with ICSE* (2003): p32-37.

[7] Li, Paul Luo, et al. Empirical evaluation of defect projection models for widely-deployed production software systems. Vol. 29. No. 6. ACM, 2004.

[8] Neufelder, A M. *Ensuring Software Reliability.* Marcel Decker, Inc. 270 Madison Avenue, New York, NY 10016, USA

[9] Savage, L.J. *The Foundations of Statistics.* Dover Publications, Inc. 180 Varick Street, New York, NY 10014, USA