Estimating The Number of Residual Defects

Yashwant K. Malaiya
Jason Denton
Computer Science Dept.
Colorado State University
Fort Collins, CO 80523
malaiya | denton@cs.colostate.edu

Abstract

Residual defects is one of the most important factors that allow one to decide if a piece of software is ready to be released. In theory, one can find all the defects and count them, however it is impossible to find all the defects within a reasonable amount of time. Estimating defect density can become difficult for high reliability software, since remaining defects can be extremely hard to test for. One possible way is to apply the exponential SRGM and thus estimate the total number of defects present at the beginning of testing. Here we show the problems with this approach and present a new approach based on software test coverage. Software test coverage directly measures the thoroughness of testing avoiding the problem of variations of test effectiveness. We apply this model to actual test data to project the residual number of defects. The results show that this method results in estimates that are more stable than the existing methods. This method is easier to understand and the convergence to the estimate can be visually observed.

1. Introduction

The estimated number of remaining defects in a piece of code is often used as an acceptance criterion. Some recent publications suggest that leading edge software development organizations typically achieve a defect density of about 2.0 defects/KLOC [1]. Some organizations now target even lower defect densities. The NASA Space Shuttle Avionics software with an estimated defect density of 0.1 defects /KLOC is regarded to be an example of what can be currently achieved by the best methods [5]. A low defect density can be quite expensive to achieve, the Space Shuttle code has been reported to have cost about $1,000 per line of code. The cost of fixing a defect later can be several orders of magnitude higher than during development, yet a program must be shipped by some deadline dictated by market considerations. This makes the estimation of defect density a very important challenge. One conceivable way of knowing the exact defect density of a program is to actually find all remaining defects. This is obviously infeasible for any commercial product. Even if resources are available, it will take a prohibitive amount of time to find all bugs in a large program [2]. Sampling based methods have been suggested for estimating the number of remaining defects. McConnell [14] has given a method that involves using two independent testing activities, perhaps by two different teams of testers. This and other sampling technique assume that faults *found* have the same testability as faults *not found*. However, in actual practice, the faults not found represent faults that are harder to find [13]. Thus such techniques are likely to yield an estimate of faults that are relatively easier to find and thus less than the true number. Fault seeding methods [14] suffer from similar problems.

It is possible to estimate the defect density based on past experience using empirical models like the Rome Lab model [7] or the model proposed by Malaiya and Denton [9]. The estimates obtained by such models can be very useful for initial planning, however these models are not expected to be accurate enough to compare with methods involving actual test data.

Another possible way to estimate the number of faults is by using the exponential SRGM. It assumes that the failure intensity is given by

$$\lambda(t) = \beta_0^E \beta_1^E e^{-\beta_1^E t} \tag{1}$$

It can be shown that the parameters β_0^E and β_1^E depend on system and test process characteristics. Specifically, β_0^E represents the total number of defects that would be eventually found. We can estimate the number of remaining defects by subtracting the number of defects found from the value of β_0^E obtained by fitting the model to the data.

We applied this technique to several data sets, figure 1 show results typical of this technique. Several things are

worth noting about this plot. First, testing continued to reveal defects even after this technique predicted that there where no remaining defects. Second, the value of β_0^E never stabilizes. Towards the end of testing β_0^E takes on a value very close to the number of defects already found, and thus provides no useful information.

An SRGM relates the number of defects found to testing time spent. In actual practice, the defect finding rate will depend on test effectiveness and can vary depending on the test input selection strategy. A software test coverage measure (like block, branch, and P-use coverage etc.) directly measures the extent to which the software under test has been exercised. Thus we expect that a suitably chosen test coverage measure will correlate better with the number of defects encountered. The relationship between test coverage and the number of defects found has been investigated by Piwowarski, Ohba and Caruso [16], Hutchins, Goradia and Ostrand [6], Malaiya et al [11], Lyu, Horgan and London [8] and Chen, Lyu and Wong [3].

In the next two sections, a model for defect density in terms of test coverage is introduced and its applicability is demonstrated using test data. Section 4 presents a two-parameter approximation of the model. Finally we present some observations on this new approach.

2. A Coverage based approach

Recently a model was presented by Malaiya et al that relates the density of residual defects with test coverage measures [11]. This model is based on the Logarithmic Poisson SRGM which has been found to have superior predictive capabilities [10]. We assume that the Logarithmic model is applicable to the total number of defects found. A justification for why actual testing often results in this model can be presented by considering the distribution of testability of the defects and the non-randomness of testing approaches [12].

Test effectiveness can vary depending on the tests selected. The effect of this variability can be removed by measuring not the testing time but the actual test coverage achieved. Using the Logarithmic Poisson model we can replace time with test coverage to obtain a new model. We assume that coverage units like branches etc. also follow a Logarithmic Poisson model with respect to testing time. The assumption is based on the facts that different branches etc. have different probabilities of getting covered just like defects.

Here we use the superscript D for defects and i = 1,2,.. for various test units like blocks, branches etc. The Logarithmic Poisson model has been shown to a good choice for an SRGM model. Its superiority over some other models of the same complexity has been shown to be statistically significant [10]. Using the Logarithmic Poisson model we can express defect coverage $C^D(t)$ as,

$$C^D(t) = \frac{\beta_0^D}{N_0^D} ln(1+\beta_1^D t), \;\; C^D(t) \leq 1 \quad (2)$$

Similarly for test unit i coverage, let us assume,

$$C^i(t) = \frac{\beta_0^i}{N_0^i} ln(1+\beta_1^i t), \;\; C^i(t) \leq 1 \quad (3)$$

where N_0^D is the total initial number of defects and N^i is the total number of units of type i in the program under test. Here $\beta_0^D, \beta_1^D, \beta_0^i, \beta_1^i$, are appropriate parameters for the applicable Logarithmic Poisson model.

We can solve for t using equation 3 and substitute it in equation 2 to obtain

$$C^D(C^i) = a_0^i ln[1+a_1^i(e^{a_2^i C^i}-1)], \;\; C^i \leq 1 \quad (4)$$

where

$$a_0^i = \frac{\beta_0^D}{N_0^D} \quad a_1^i = \frac{\beta_1^D}{\beta_1^i} \quad a_2^i = \frac{N^i}{\beta_0^i} \quad (5)$$

Often we want to use the number of defects found rather than defect coverage. If we indicate the total expected number of defects found in time t by $\mu(t)$, we can write $C^D(t) = \frac{\mu(t)}{N_0^D}$. Hence from equation 4,

$$\mu^D(C^i) = a_3^i ln[1+a_1^i(e^{a_2^i c^i}-1)], \;\; C^i \leq 1 \quad (6)$$

where $a_3^i = a_0^i \times N_0^D = \beta_0^D$

Equation 6 can be used when the initial number of defects is not available. We must note that equations 4 and 6 are applicable only when C^i is less than or equal to one. Note that equation 6 allows for less than 100% defect coverage when enumerable coverage is at 100%. This is because exercising all branches does not exercise the program exhaustively. Achieving a 100% coverage using a more strict measure, such as P-uses, would exercise the code more thoroughly.

Figure 2 shows a plot illustrating the shape of the curve described by equation 4. At the beginning, defect coverage grows only slowly with test unit coverage. However, at higher test coverage, there is a linear relationship. The value around which the curve exhibits a knee has a significance as we will see below. The important point is to note that at 100% coverage of unit i, we expect to find the number of defects given by the point where the curve intersects the vertical line corresponding to 100% coverage. We can expect that at least this many faults were initially present. A lower bound on the number of remaining faults is obtained by subtracting the number of faults actually found from the number of faults expeceted at 100% coverage. The bound is closer to the actual number if we use a more strict test coverage measure.

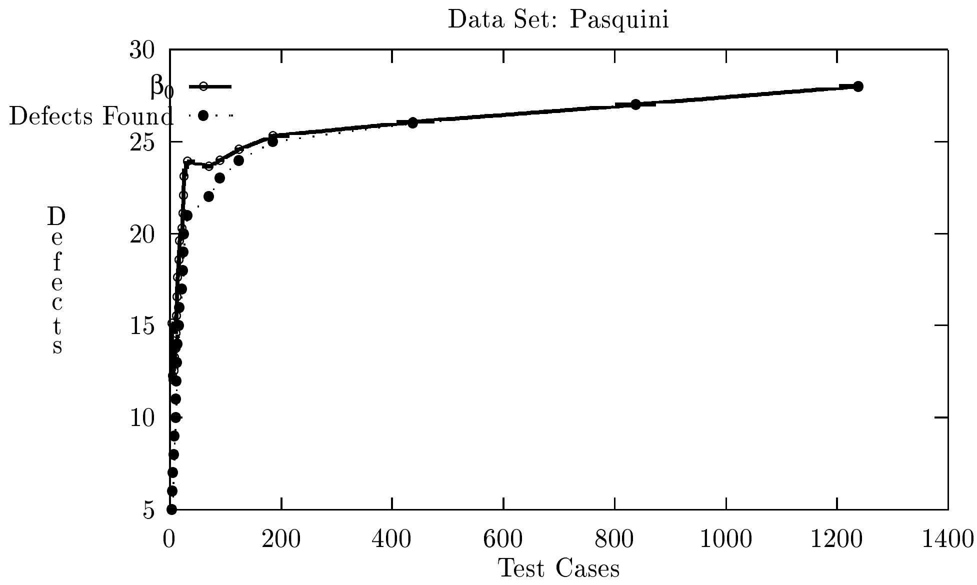

Figure 1. Estimated total defects using exponential model (Pasquini data)

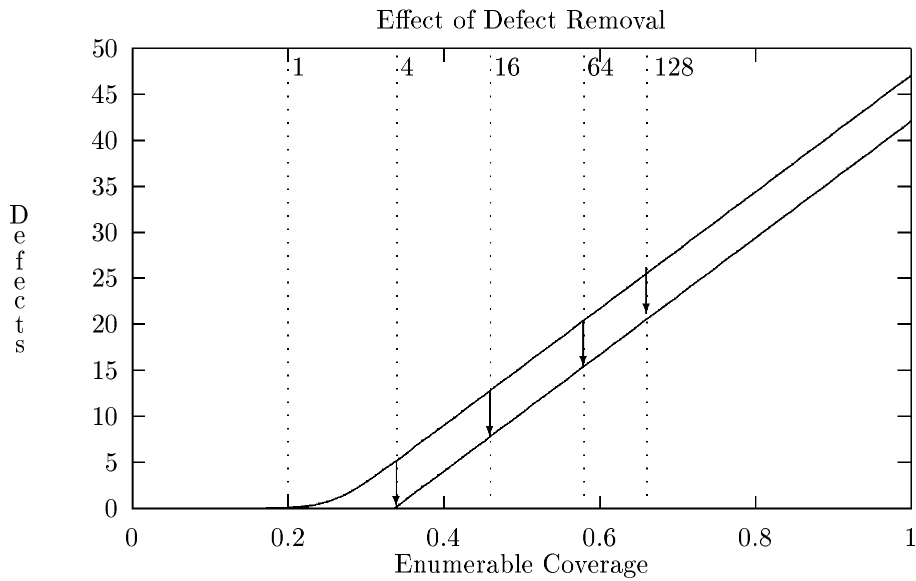

Figure 2. Defects vs. Test Coverage Model

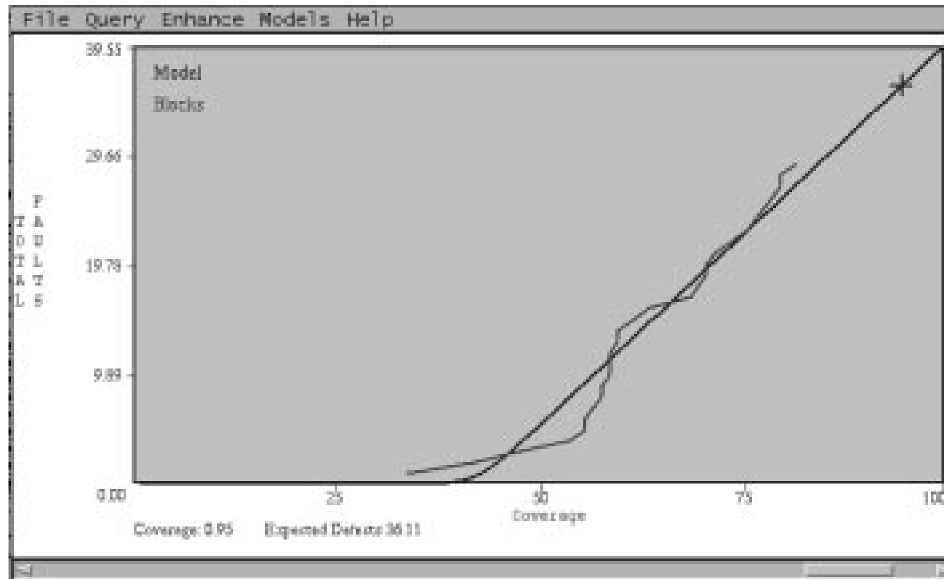

Figure 3. ROBUST: Block Coverage data (Pasquini et al.)

The numbers across the top of the plot in Figure 2 give the number of test cases applied for the top curve. They show that despite the linear growth of defect coverage, more testing effort is needed to obtain more coverage and thus find more faults. The second curve shows the plot that would be obtained if some of faults were already removed in a previous test phase when current testing was initiated. The second plot shows that when the initial defect density is lower, testing starts finding defects at a higher coverage level. This corresponds to to shifting of the curve downwards.

3. Applying the New Approach

Applicability of this model is illustrated by the plots in figures 3, 4, and 5. This data was collected experimentally by Pasquini et al from a 6100 line C program by applying 20,000 test cases [15] . The test coverage data was collected using the ATAC tool. Figure 3 shows a screen from ROBUST, a tool which implements our technqiue; and that has been developed at CSU [4]. Further development of this tool is underway to include additional capabilities.

For the 20,000 tests, the coverage values obtained were: block coverage : 82.31% of 2970 blocks, decision coverage : 70.71% of 1171 decisions, and p-use coverage 61.51% of 2546 p-uses. This is to be expected since p-use coverage is the most rigorous coverage measure and block coverage is the least. Complete branch coverage guarantees complete block coverage, and complete p-use coverage guarantees complete decision coverage.

The plots suggest that 100% block coverage would uncover 40 defects, 100% branch coverage would uncover 47 defects, whereas 100% p-use coverage would reveal 51 defects. If we assume that all of the software components are reachable, than we can expect that the actual total number of faults to be slightly more than 51. In actual practice, often large programs contain some unreachable code, often called dead or obsolete code. For C programs it can be in the neighborhood of 5%. For accurate estimates this needs to be taken into account. For simplicity of illustration, here we will assume that all of the code is reachable. Unreachable code can be minimized by using coverage tools and making sure that all modules and sections are entered during testing.

The data sets used in this paper are for programs that are not evolving and thus the actual defect density is constant.

We note that the fitted model becomes very linear after the knee in the curve. Let us define C_k^i as the knee, where the linear part intersects the x-axis. For block, branch and p-use coverage it occurs at about 40%, 25% and 25% respectively. Below we see the significance of this value.

The coverage model in Equations 4 and 6 provides us a new way to estimate the total number of defects. As we can see in Figures 3, 4 and 5, which use the data obtained by Pasquini et al., the data points follow the linear part of the model rather closely. Both the experimental data and the model suggest that the 100% coverage eventually achieved should uncover the number of faults as given in Table 1 below. Numbers in the last column have been rounded to nearest integer.

It should be noted that for this project 1240 tests revealed 28 faults. Another 18,760 tests did not find any additional faults, even though at least 5 more faults were known. The data suggests that the enumerables (blocks, branches etc.)

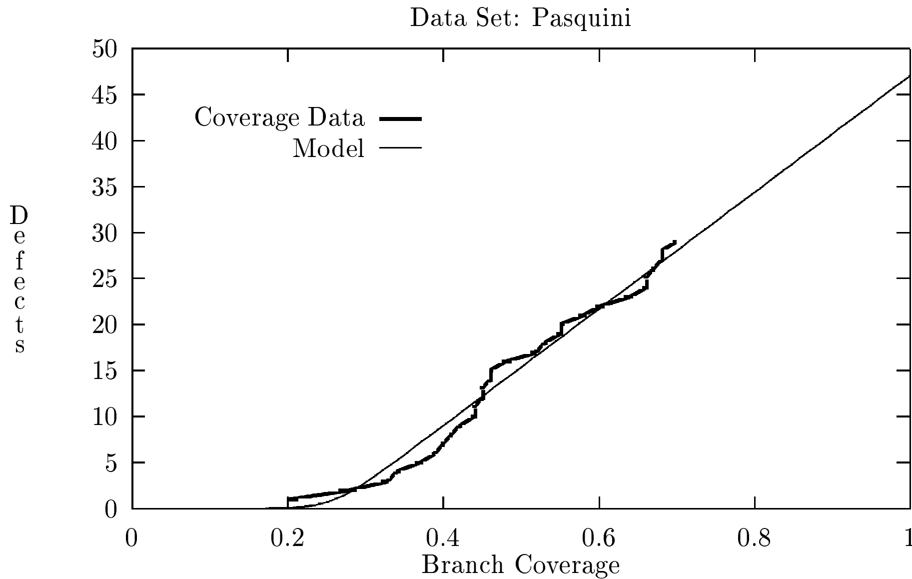

Figure 4. Defects vs. % Branch Coverage

Table 1. Projected number of total defects with 100% coverage

Coverage Measure	Defects found	Coverage achieved	Defects expected
Block Coverage	28	82%	40
Branch Coverage	28	70%	47
P-uses Coverage	28	67%	51
C-uses Coverage	28	74%	43

not covered by the first 1240 tests were very hard to reach. They perhaps belong to sections of the code intended for handling special situations.

There is a subsumption relationship among blocks, branches and P-uses. Covering all P-uses assures covering all blocks and branches. Covering all branches assures coverage of all blocks. Thus among the three measures the P-use coverage measure is most strict. There is no coverage measure such that 100% coverage will assure detection of all the defects. Thus in the above table, the entry in the last column is a low estimate of the total number of defects actually present. Using a more strict coverage measure raises the low estimate closer to the actual value. Thus the estimate of 51 faults using P-use coverage should be closer to the actual number than the estimates provided by block coverage. Two coverage measures, DU-path coverage and all-path coverage are more strict than P-use coverage, and may be suitable for cases where ultra-high reliability is required. However considering the fact that often even obtaining 100% branch coverage is infeasible, we are unlikely to detect more faults than what the P-use coverage data provides even with fairly rigorous testing. It should be noted that C-use coverage does not fit in the subsumption hierarchy and therefore it is hard to interpret the values obtained by using C-use coverage data.

Further application of this new method is illustrated by examining the data provided by Vouk [17]. These three data sets were obtained by testing three separate implementations of a sensor management program for an inertial navigation system. Each program is about five thousand lines of code. In the first program, 1196 tests found 9 defects. For the other two programs 796 test revealed 7 and 9 defects respectively. Figure 6 shows the plots of P-use coverage achieved versus the number of defects found. Table 2 shows the estimates for the total number of faults that would be found with 100% coverage.

Table 2. Projected number of total defects with 100% coverage (Vouk's data sets)

Data Set	Faults Found	Expected Faults		
		Block	Branch	P-use
Vouk1	9	11	11	18
Vouk2	7	8	8	8
Vouk3	10	10	11	13

Table 2 again shows that the estimates obtained are consistent with the subsumption hierarchy.

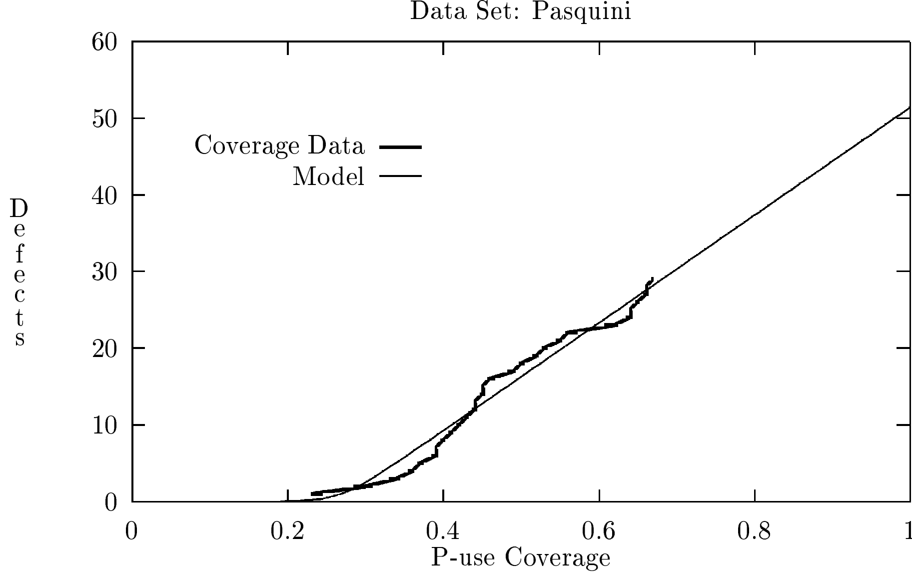

Figure 5. Defects vs. % P-use Coverage

4. Significance of Parameters

Figure 2, which is a direct plot of the model, and figures 3, 4, and 5 which plot actual d ata, suggest that at higher coverage values, a linear model can be a very good approximation. This leads us to an analytical interpretation of the behavior. This interpretation can be used for obtaining a simple preliminary model for planning purposes.

We will obtain a linear model from equation 4. Let us assume that at higher values of C^i equation 4 can be simplified as

$$\begin{aligned} C^D(C^i) &= a_0^i ln[a_1^i e^{a_2^i C^i} \\ &= a_0^i ln(a_1^i) + a_0^i a_2^i C^i \\ &= A_0^i + A_1^i C^i \text{ where } C^i > C_n^i \end{aligned} \tag{7}$$

$$A_0^i = a_0^i ln(a_1^i) = \frac{\beta_0^D}{N_0^D} ln\left(\frac{\beta_1^D}{\beta_1^i}\right) \tag{8}$$

and

$$A_1^i = a_0^i a_1^i = \frac{\beta_0^D}{N_0^D}\frac{N^i}{\beta_0^i} \tag{9}$$

Note that this simplification is applicable only for values of C_n^i greater than where the knee occurs. The experimental parameters values for this model can not be obtained until a clear linear behavior beyond the knee has been established. Assuming that the knee occurs where the linear part of the model intersects the x-axis, using equation 7, the knee is at

$$C_{knee}^i = -\frac{A_0^i}{A_1^i} \tag{10}$$

Here we can make a useful approximation. For a strict coverage measure, for $C^i = 1, C^D \approx 1$. Hence from equation 7, we have

$$A_0^i + A_1^i \approx 1. \tag{11}$$

Replacing A_0^i by $1 - A_1^i$ in equation 10, and using 8, we can write,

$$C_{knee}^i = 1 - \frac{1}{a_0^i a_1^i} \tag{12}$$

This can be written as [9],

$$C_{knee}^i = 1 - (\frac{D_{min}^i}{D_{min}^0 D_0^i}) D_0^0 \tag{13}$$

Where D_{min}^i, D_{min}^0, D_0^i are parameters and D_0^0 is the initial defect density. Thus for lower defect densities, the knee occurs at higher test coverage. This has a simple physical interpretation. If a program has been previously tested resulting in a lower defect density, it is likely that the enumerables with higher testability have already been exercised. This means that testing will start finding a significant number of additional defects only after higher test coverage values are achieved.

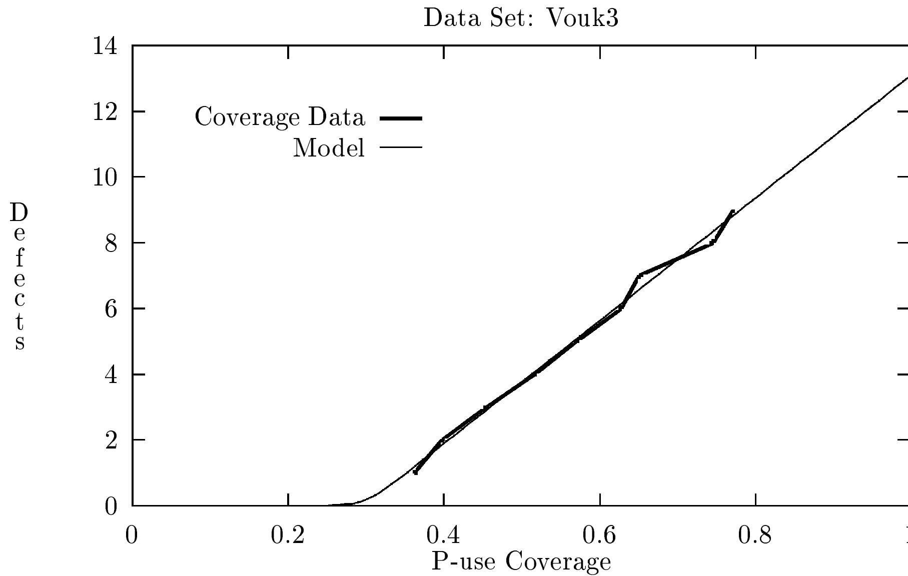

Figure 6. Defects vs. % P-use Coverage

5. Conclusions

Defect density is an important measure of software reliability, figuring prominently in the reliability assessment of many quality assurance engineers and managers. Existing methods for estimating the number of defects can underestimate the number of defects because of a bias towards easily testable faults. The exponential model tends to generate unstable projections and often yield a number equal to defects already found.

We have presented a model for defects found based on coverage, and shown that this model provides a very good description of actual data. Our method provides stable projections early in the development processes. Experimental data suggests that our method can provide more accurate estimates and provide developers with the information they require to make an accurate assessment of software reliability. The choice of coverage measure does has an effect on the projections made, and we suggest that a strict coverage measure such as P-uses should be used for very high reliability programs.

Here we have assume that no new defects are being introduced in the program. Further investigations are needed to extend this method for evolving programs.

6. Acknowledgement

This work was supported in part by a BMDO funded project monitored by ONR.

References

[1] R. V. Binder. Six sigma: Hardware si, software no! *http://www.rbsc.com/pages/sixsig.html*, 1997.

[2] R. W. Butler and G. B. Finelli. The infeasibility of quantifying the reliability of life-critical real-time software. *IEEE Transactions on Software Engineering*, 19(1):3–12, Jan. 1993.

[3] M. Chen, M. R. Lyu, and W. E. Wong. An empirical study of the correlation between coverage and reliability estimation. In *Proc. IEEE Third International Symposium on Software Metrics*, pages 133–141, Berlin, Germany, Mar. 1996.

[4] J. A. Denton. *ROBUST, An Integrated Software Reliability Tool*. Colorado State University, 1997.

[5] L. Hatton. N-version design versus one good design. *IEEE Software*, pages 71–76, Nov./Dec. 1997.

[6] M. Hutchings, T. Goradia, and T. Ostrand. Experiments on the effectiveness of data-flow and control-flow based test data adequacy criteria. In *Proc. International Conference on Software Engineering*, pages 191–200, 1994.

[7] P. Lakey and A. Neufelder. *System and Software Reliability Assurance Notebook*. Rome Laboratory, Rome, New York, 1997.

[8] M. R. Lyu, J. R. Horgan, and S. London. A coverage analysis tool for the effectiveness of software testing. In *Proc. of the IEEE International Symposium on Software Reliability Engineering*, pages 25–34, 1993.

[9] Y. K. Malaiya and J. A. Denton. What do software reliability parameters represent? In *Proc. International Symposium on Software Reliability Engineering*, pages 124–135, Albuquerque, NM, Nov. 1997.

[10] Y. K. Malaiya, N. Karunanithi, and P. Verma. Predictability of software reliability models. *IEEE Transactions on Reliability*, pages 539–546, Dec. 1992.

[11] Y. K. .Malaiya, N. Li, J. Bieman, R. Karcich, and B. Skibbe. The relationship between test coverage and reliability. In *Proc. of the International Symposium on Software Reliability Engineering*, pages 186–195, Nov. 1994.

[12] Y. K. Malaiya, A. von Mayrhauser, and P. Srimani. An examination of fault exposure ratio. *IEEE Transactions on Software Engineering*, pages 1087–1094, Nov. 1993.

[13] Y. K. Malaiya and S. Yang. The coverage problem for random testing. In *Proc. IEEE International Test Conference*, pages 237–245, Oct. 1984.

[14] S. McConnell. Gauging software readiness with defect tracking. *IEEE Software*, 14(3), May/June 1997.

[15] A. Pasquini, A. N. Crespo, and P. Matrella. Sensitivity of reliability growth models to operational profile errors. *IEEE Transactions on Reliability*, pages 531–540, Dec. 1996.

[16] P. Piwowarski, M. Ohba, and J. Caruso. Coverage measurement experince during function test. In *Proc. of the 15th International Conference on Software Engineering*, pages 287–300, May 1993.

[17] M. A. Vouk. Using reliability models during testing with non-operational profiles. In *Proc. of the 2nd Bellcore/Purdue Workshop on Issues in Software Reliability Estimation*, pages 103–111, Oct. 1992.