14

Poisson Models for Subprogram Defect Analyses

Dr. William M. Evanco
The MITRE Corporation
7525 Colshire Drive
McLean, Virginia 22101-3481
e-mail: evanco@mitre.org
Fax: (202) 863-2970

Abstract

For Ada systems, the hierarchy of subprograms compose a layered virtual machine within an object-based framework. Poisson analyses are proposed for the identification of the determinants of defects in these subprograms. Software complexity (measured during design or implementation) as well as characteristics of the software development environment influence the number of defects identified during the testing phase. The Poisson models are calibrated on the basis of measures extracted from the code of Ada projects and data from software change reports. One of the models is used to estimate defects at the subprogram and project levels for the calibration data, and these estimates are then compared to the actual defects. To demonstrate cross-language applicability, defect predictions are made at the subsystem level for a project coded in the C programming language and compared to the actual subsystem defects. Notable results from the analysis are that extensively modified reused subprograms (>25% changed) have substantially more defects than new code of otherwise comparable characteristics and that software development environment volatility (as measured by non-defect changes per thousand source lines of code) is a strong determinant of subprogram defects.

Keywords

Software metrics, software quality, multivariate analysis, prediction models, reliability modeling, Poisson analysis

1 INTRODUCTION

A wide variety of metrics characterizing software and the software development process have been identified in the research literature (for reviews, see Cote et al., 1988; Zuse, 1990). Of particular interest are measures of software complexity, many of which have been related to software quality factors such as reliability or maintainability (Agresti and Evanco, 1992; Agresti et al., 1990; Kafura and Reddy, 1987).

Most of these software complexity metrics can be collected relatively early in the development life cycle (e.g., during design or implementation). On the other hand, indicators of software quality generally emerge later in the life cycle (e.g., during testing or operation). Thus, software complexity measures provide early indications of software quality.

S. Bologna et al. (eds.), *Achieving Quality in Software*

However, as has been pointed out by a number of authors, no single metric can adequately capture the complexity of software (Basili and Rombach, 1988; Evanco and Agresti, 1992; Munson and Khoshgoftaar, 1992; Selby and Porter, 1988). Rather, software complexity can be characterized along different dimensions, each dimension being measured by at least one metric. The problem then becomes one of deciding how to integrate the metrics to determine their joint contributions to software quality. Methodologies to accomplish this integration of metrics are a topic of current research interest.

1.1 Discriminant Analysis

For example, Munson and Khoshgoftaar (1992) propose that many of the available metrics can be clustered into correlated groups. Each group represents a domain of software complexity such as control flow or data structure complexity. A principal components analysis identifies these groups from which a smaller number of orthogonal domain metrics can be defined. Each domain metric has an associated eigenvalue representing the relative amount of variance explained by the domain. Multiplying the domain metrics by their eigenvalues and summing the products provides a unitary metric for the relative complexity of a software component. This relative complexity may then be used to identify, for example, defect-prone software modules.

Selby and Porter (1988) propose a classification tree methodology to identify defect prone or difficult-to-maintain modules. A tree generation algorithm produces a classification tree using various software metrics for a group of previously developed modules. The classification tree is then applied to a new set of modules in order to identify potentially troublesome ones based on their metric characteristics.

Both of these approaches rely on some form of discriminant analysis. For example, an integer value is selected as a cutoff and modules with defect numbers exceeding this value are regarded as defect prone. Unfortunately, these methodologies do not provide quantitative estimates of the numbers of module defects.

The argument made for the use of discriminant analysis is its appropriateness for analyzing relatively small software components such as subprograms typically having few defects. In fact, a substantial number of subprograms may have zero defects, yielding a defect distribution skewed toward zero. Additionally, defects are measured on an integer scale and for small defect numbers the integer scale cannot be adequately approximated by a continuous one. These conditions rule out the use of a statistical technique such as ordinary least squares, which requires that the dependent variable (i.e., defects) be continuous and normally distributed.

1.2 Quantitative Defect Analysis

There is a need, nevertheless, for an approach that integrates a variety of software metrics into models to provide a quantitative measure of software defects for small software components. Such methodologies have a clear advantage over discriminant analyses that only provide classification capabilities. For example, during the design phase, defect prediction models may be used to evaluate the impact of design changes on defect numbers. Prior to testing, components at risk of being under-tested could be identified by comparing the planned test coverage with the predicted defect distribution. During testing, additional test cases may be identified by comparing the actual defects revealed through testing with the predicted defects. And, finally, a decision to stop testing could be made when the actual defect numbers approach the predicted defects.

One such technique for integrating metrics was proposed by Evanco and Agresti (1992). Using an ordered response model, a composite complexity measure was calibrated. This composite complexity measure, expressed as a linear combination of basic software complexity

metrics (e.g., calls per subprogram) and measures of the development environment complexity (e.g., non-defect changes), is an order statistic. Assuming a probability distribution for the composite complexity, probabilities were estimated for membership in one of the n+2 categories: no defects, one defect, two defects,..., n defects, and greater than n defects, where n is a positive integer chosen so that few modules have more than n defects. These probabilities were then used to calculate the expected number of defects at the library unit aggregation level for Ada programs.

In this study, another approach based on Poisson analysis is presented. Poisson models treat the defect number as a random variable that can assume any non-negative integer value. The Poisson methodology is applied to analyze the determinants of defects in Ada subprograms.

1.3 Subprograms in Ada Systems

Software analyses have been conducted at the level of physical modules such as subprograms (Kafura and Reddy, 1987) in FORTRAN or C, or some rollup of physical modules such as library unit aggregations (Evanco and Agresti, 1992), subsystems (Agresti and Evanco, 1992; Agresti et al., 1990), or projects (Card and Agresti, 1988). However, within the Ada programming language, a subprogram is not the natural physical unit of encapsulation. The compilation unit plays this role and may encapsulate a package specification or body, a subunit, and, sometimes, library unit subprograms. Typically, though, subprograms appear as program units encapsulated within package library units and such subprograms may be regarded as submodules.

The hierarchy of subprograms (expressed as a call tree) can be viewed as a layered virtual machine (LVM). Nielson and Shumate (1988) combine the concepts of the LVM and object-oriented design (OOD) into a design methodology whereby a software system is decomposed into a hierarchy of virtual machines (i.e., Ada subprograms) and objects (e.g., Ada packages, types, and objects of the types). While the version of Ada, namely Ada83, considered in this study does not fully support the object-oriented paradigm, it nevertheless has some features of an object-oriented language. Ada83 allows for information hiding through its packaging constructs to ensure the reliability and modifiability of software by controlling dependencies. Ada83 also supports data abstraction through abstract data types. Ada95 is, however, a fully object-oriented language supporting in addition, dynamic binding and inheritance. Because Ada83 does not support all of the features of an object-oriented language it is sometimes referred to as an *object-based* language.

In previous work (Agresti and Evanco, 1992; Agresti et al., 1990 ; Evanco and Agresti, 1992), we focused on some of the object-based features of Ada83 (e.g., context coupling of packages) expected to influence defects. In this study, our attention is turned to the analysis of the characteristics of the LVM that may affect defects. Future work will focus on merging the LVM and the object-based features into an integrated framework.

For Ada systems, subprogram level analyses have a number of advantages over analyses of library unit aggregations. A library unit aggregation, encapsulating multiple subprograms, averages about 650 source lines of code for the Ada programs used in this study. On the other hand, a subprogram averages about 100 source lines of code. Therefore, the identification of defects can be better localized for subprograms.

Second, subprograms rather than packages may be a more natural unit for analysis with respect to the testing process. A functional profile can be derived when testing is conducted according to some operational profile (Musa, 1993). The functional profile identifies those functional capabilities of interest and value to end-users along with their usage probabilities. Since subprograms embody the functionality of the system, the ability to predict subprogram

defects provides a means for monitoring and controlling the progress of testing. Actual defects may be compared to predicted defects in order to identify potentially under-tested subprograms.

Third, the Ada subprogram level analyses in this study can be extended to procedural languages such as FORTRAN or C. The explanatory variables for the Ada subprogram defect prediction models discussed below are generic to procedural languages since no reference is made to Ada-specific constructs. Thus, subprogram level analyses enable us to handle systems with mixed programming languages such as Ada systems with calls to FORTRAN or C subprograms.

Finally, subprogram level analyses can facilitate the reengineering of a system from a procedural language such as FORTRAN or C to the object-based language of Ada. Measurements of the system can be made at each stage of the reengineering process. Reverse engineering of the procedural language system provides "views" from which metrics can be derived to evaluate software components for restructuring. For example, the call tree and logic control flow views are characterized by fan-out and cyclomatic complexity metrics, respectively. These metrics can help to identify excessively complex components for restructuring. Similarly, measurements of the reengineered system allow comparisons to the original system to evaluate whether or not it is an improvement.

In the next section, the Poisson methodology is presented and the determinants of subprogram defects used in this study are discussed. In Section 3, we discuss the characteristics of the empirical data. In Section 4, the empirical results are presented. Section 5 demonstrates the cross-language applicability of the LVM results by predicting defects for a C language system. Finally, Section 6 presents the conclusions.

2 POISSON DEFECT MODEL

In order to conduct subprogram level analyses, we use a Poisson model* (Evanco and Lacovara, 1994). As discussed previously, since subprograms are relatively small units, the discrete nature of the defect number becomes apparent. Thus, least squares regression analyses, relying on assumptions of normality and continuity of the dependent variable, are ruled out. Instead, the defect number can be represented by a Poisson distribution given by

$$P(\text{Defects}=r_i;\ i) = \exp(-\lambda_i)\frac{(\lambda_i)^{r_i}}{r_i!} \tag{1}$$

where $P(\text{Defects}=r_i;\ i)$ is the probability of r_i defects (r_i=0,1, 2,...) and λ_i is the expected number of defects for the ith subprogram. λ_i is a non-negative function of both the complexity characteristics of the ith subprogram and the development environment characteristics that can be expressed in log-linear form as:

$$\ln(\lambda_i) = a_0 + a_1*\ln(1+NSC) + a_2*\ln(1+NP) + a_3*\ln(CC) + a_4*\ln(1+ND) + a_5*\ln(1+NCH) + a_6*NCI + a_7*EMC \tag{2}$$

---

* Non-homogeneous Poisson models have been used previously for reliability growth analyses (Goel and Okumoto, 1979). In such models, the parameter is a function of time and the units of observation typically have been projects. In this study, the subprogram is the unit of observation and the parameter is a function of subprogram characteristics. The number of defects is the total number accumulated throughout the testing phase.

where "ln" is the natural logarithm and a_0, a_1,..., a_m are parameters to be estimated. The explanatory variables are defined as:

- NSC = number of subprogram calls
- NP = number of parameters
- CC = cyclomatic complexity
- ND = average nesting depth
- NCH = non-defect changes per source line of code
- NCI = new code indicator (equals one if subprogram is completely new code and zero otherwise)
- EMC = extensively modified reused code indicator (equals one if subprogram is reused but extensively modified and zero otherwise).

The first four variables represent characteristics of the software design/code. These characteristics are not Ada-specific and may also be obtained for procedural languages such as FORTRAN or C. The *number of subprogram calls* and the *number of parameters* may be derived through the analysis of system design documentation. Using design documentation to determine these complexity attributes provides useful information about potential determinants of defects early in system development. On the other hand, the *cyclomatic complexity* and the *average nesting depth* may be available late in the design stage but generally are available at some point in the implementation phase.

One way of characterizing the structural complexity of a software system is by means of a subprogram call graph. The nodes of the graph represent subprograms and the directed arcs represent subprogram calls. The *number of subprogram calls* is a measure of the contribution by a subprogram to the structural complexity of a software system. This measure is an indicator of a subprogram's connections to its external environment through calls to other subprograms and is a major component of the fan-out measure introduced by Henry and Kafura (1981). We hypothesize that increasing a subprogram's external complexity will tend to increase its defects.

Parameters are data items that are manipulated by the executable code of subprograms. These parameters serve as inputs to or outputs from other subprograms. The process of mapping from a set of inputs to a set of outputs may be interpreted as the computational workload of a subprogram. Adding parameters to a subprogram tends to increase its computational workload and, hence, its internal complexity, leading to potentially more defects. Therefore, the *number of parameters* is an indicator of the workload performed by a subprogram (Card and Agresti, 1988). We establish the convention that parameters which are data structures such as vectors or matrices increment the parameter count by one.

The McCabe *cyclomatic complexity* has been proposed in the literature as an important determinant of defects (McCabe, 1976; Walsh, 1979). It is a measure of a subprogram's internal complexity from the perspective of logic control flows. A directed graph can be derived to represent these control flows. The cyclomatic complexity is the number of independent paths in the directed graph. Equivalently, the cyclomatic complexity can be calculated by counting the simple Boolean conditions in the control statements (e.g., while statements, do loops, etc.). We hypothesize that the number of subprogram defects tends to increase as the use of control statements to implement a subprogram's workload increases. The cyclomatic complexity measure generally becomes available later in the development process when a subprogram body is implemented in terms of its control flow logic.

The *average nesting depth* is the final indicator of the internal complexity of a subprogram. This measure complements the cyclomatic complexity measure which varies with the number of program predicates but is not sensitive to the complexity associated with nesting structures. The average nesting depth measure is explicitly concerned with program nesting structures and their contribution to complexity. To compute the average nesting depth, each statement of the

subprogram is assigned a nesting depth. This depth is incremented by one when entering or decremented by one when leaving, for example, a block declarative region, a begin-end block, a logic control flow block (e.g., loop, case, or if statements), and exception handler alternatives. The average nesting depth is obtained by summing the nesting depths of the statements and dividing by the total number of statements.

The remaining variables represent features of the development environment. These variables are not collected by means of a software analyzer, but rather through additional data provided by the developer about software changes and the reuse of software components.

The *number of non-defect changes per source line of code* made to a subprogram is an indicator of development environment volatility. This variable has been shown to influence subsystem level defect densities in a previous study (Agresti and Evanco, 1992). The non-defect changes may be a result of new or unanticipated requirements that emerge during the development process. New requirements may be poorly communicated or inadequately understood, leading to implementation defects. In addition, the new requirements may be difficult to implement in the framework of the original design since these requirements were not initially anticipated. In any case, the resulting changes contribute to development complexity and we hypothesize that the changes may lead to additional defects.

The ability of an organization to reuse previously developed code is expected to influence the number of defects. Code that is reused verbatim (without any modifications) exhibits very few defects in our data (14 defects in about 50,000 source lines of code). Therefore, we restricted the analyses to new code, reused but slightly modified code, and reused but extensively modified code.

The *new code indicator,* NCI, is a dummy variable equal to unity for a subprogram consisting of new code and zero otherwise. Similarly, the *reused but extensively modified code indicator,* EMC, equals unity for a subprogram developed from reused code with extensive modifications and zero otherwise. If more than 25% of a subprogram's code has been changed, then it is regarded as having been extensively modified. Thus, the (NCI,EMC) values are (1,0) for new code, (0,1) for extensively modified code, and (0,0) for slightly modified code (i.e., less than 25% modified). The effect of the dummy variables, NCI and EMC, is to shift the constant term, a_0, of equation (2) by a_6 and a_7 respectively. Our working hypothesis is that slightly modified reused code will have fewer defects than either new code or extensively modified reused code. Thus, we expect a_6 and a_7 to have positive values.

From the probabilities in equation (1), the values of the coefficients, a_j, can be estimated using a maximum likelihood approach. Given N empirical observations, the likelihood function is expressed as:

$$L(a_0, a_1, ..., a_7) = \prod_{i=1}^{N} \exp(-\lambda_i) \frac{(\lambda_i)^{r_i}}{r_i!} \qquad (3)$$

where the λ_i are functions of the parameters, $a_0, a_1, ..., a_7$, as indicated in equation (2). Taking the derivatives of (3) with respect to the parameters, setting the derivatives equal to zero, and solving for the parameters yields solutions for the parameters optimizing equation (3).

3 EMPIRICAL DATA

The subprograms were obtained from four flight dynamic and telemetry simulation programs written in Ada83 by the NASA Software Engineering Laboratory (NASA/SEL). These

programs were analyzed using the Ada Static Source Code Analyzer Program (ASAP) (Doubleday, 1987). The raw outputs of ASAP were input to a number of tools and utilities to extract subprogram level data. The subprograms include both library units and program units contained within library units.

A total of 1013 new, reused but slightly modified, and reused but extensively modified subprograms were obtained from the four projects. Reused subprograms with no modifications were not considered since they contain very few defects. About 81% of the code is new while 19% is reused with some modifications. About 6% of the subprograms are library units while the rest are program units contained within library units. About 14% of the subprograms are at the bottom of the call tree, making no calls to other subprograms.

Change report data were tabulated for defects and for non-defect changes and component origination forms were analyzed to extract data on software reuse. Additional summary statistics for the subprogram data are shown in Table 1.

Table 1 Characteristics of Subprogram Data

Variable	Mean	Standard Deviation	Minimum	Maximum
Defects	1.02	1.54	0	11
Source lines of code	93	97	5	927
Subprogram calls	11.5	18.2	0	142
Parameters	2.7	3.3	0	48
Cyclomatic complexity	23.3	29.8	1	249
Average nesting level	1.6	.77	0	6
Non-defect changes	1.5	1.8	0	18

4 ANALYSIS RESULTS

Estimates of the parameters in equation (2) are shown in Table 2 for three different models. The standard deviations of the parameter estimates are the numbers in parentheses. The first column gives the names of the variables associated with the parameters. The second column shows the analysis results for Model A which involves only those software complexity metrics that are expected to be available at design time: the number of subprogram calls and the number of parameters. Model B, shown in the third column, incorporates the additional software complexity measures available during implementation: the cyclomatic complexity and the average nesting level but excludes the number of parameters count. Finally Model C includes all of the variables of Model B plus the number of parameters count. In the three models, all of the variables are at least at the 5% level of significance.

The last row of Table 2 shows the correlations of the predicted defects calculated from equation (2) with the actual defects for the different models. Model A has a correlation coefficient of .41. The correlation coefficient of Model B is .46, while that of Model C is .48.

Table 2 Poisson Model Estimates of Subprogram Defects

Variable[1]	Model A	Model B	Model C
Intercept	-1.63 (.14)	-2.17 (.16)	-2.28 (.16)
Subprogram Calls[2]	.48 (.03)	.25 (.04)	.26 (.04)
Number of Parameters[2]	.14 (.04)		.11 (.05)
Cyclomatic Complexity		.27 (.04)	.27 (.04)
Average Nesting Depth[2]		.47 (.16)	.39 (.16)
Non-Defect Changes/ Source Lines of Code[2]	4.8 (1.3)	5.6 (1.4)	5.7 (1.4)
New Code Indicator	.41 (.11)	.34 (.11)	.37 (.11)
Extensively Modified Code Indicator	.73 (.16)	.62 (.16)	.67 (.16)
Coefficient of Correlation	.41	.46	.48

[1] All variables are entered in logarithmic form with the exception of the indicators for new and extensively modified code.

[2] One is added to the value of the variable to prevent zero-valued arguments in the logarithm.

Taking the differential of equation (2) with respect to one of the software complexity variables, X, on the right hand side yields:

$$\frac{\Delta\lambda}{\lambda} = a_j * \frac{\Delta X}{X} \qquad (4)$$

Since λ is the expected number of defects, the parameter a_j is interpreted as the *elasticity* of the expected number of defects with respect to the corresponding variable, X. For example, in Model C a 10% decrease in cyclomatic complexity leads to a 2.7% decrease in defects.

The elasticities associated with the four software complexity measures are all substantially less than unity. Taking the anti-logarithm of equation (2), the contribution of subprogram calls to predicted defects is given by $(1+NSC)^{a_1}$ where from Table 2, a_1 equals .48 for Model A, .25 for Model B, and .26 for Model C. These empirical results do not support assumptions made in other studies (Card and Agresti, 1988; Henry and Kafura, 1981) that the number of defects varies as the square of subprogram calls.

Most notable in Table 2 is the very high elasticity associated with development environment volatility as measured by non-defect changes per source line of code. For example, in Model C, a 10% increase in (1+NCH) increases defects by 57%.

The results in Table 2 also indicate that reused slightly modified code (NCI=0, EMC=0) exhibits the fewest defects since the parameters associated with NCI and EMC are positive. Reused extensively modified code (NCI=0, EMC=1) has more defects than new code (NCI=1, EMC=0), since the parameter for EMC is greater than the parameter for NCI.

Taking reused slightly modified code as a baseline, in Model C new code is expected to have about 45% more defects ($e^{.37}=1.45$), while extensively modified reused code is expected to have about 95% more defects ($e^{.67}=1.95$). Thus, if a potentially reusable subprogram requires that more than 25% of its code be modified then it may be better to develop new code if the objective is to minimize defects. However, tradeoff analyses are needed to compare the costs of new development with the costs of extensively modifying reused code and correcting the additional defects.

These results for reuse are related to the large impact that non-defect code changes have on defects. Adapting code for reuse in a new system involves non-defect changes (however, these changes are not included in the counts of non-defect changes per source line of code). The analysis results for reuse and for non-defect changes support the view that adapting code to account for new or modified requirements is an important risk factor contributing to software defects.

In addition to the variables shown in Table 2, we also examined the impact on expected defect numbers of a subprogram being either a library unit or program unit. A dummy variable was introduced equal to unity for a library unit and zero for a program unit. The associated coefficient was small and statistically insignificant, indicating that this distinction was not an important factor in determining defects.

The predicted and actual defects for the four projects in this analysis were rolled up to the subsystem level to graphically demonstrate the fit of the model. These rollups are plotted in Figure 1 for Model C. Perfect predictions would lie on the forty-five degree line. The actual and predicted defects fit well with the exception of one outlier. The correlation between actual and predicted defects is .93.

Project level rollups for the four projects used in the analysis are shown in Table 3. The predicted defects and the defect densities (expressed as defects per thousand source lines of code) compare well to the actuals except for Project D. This project involves mostly verbatim reuse and the remaining subprograms that were used in this analysis constitute only 2800 source lines of code.

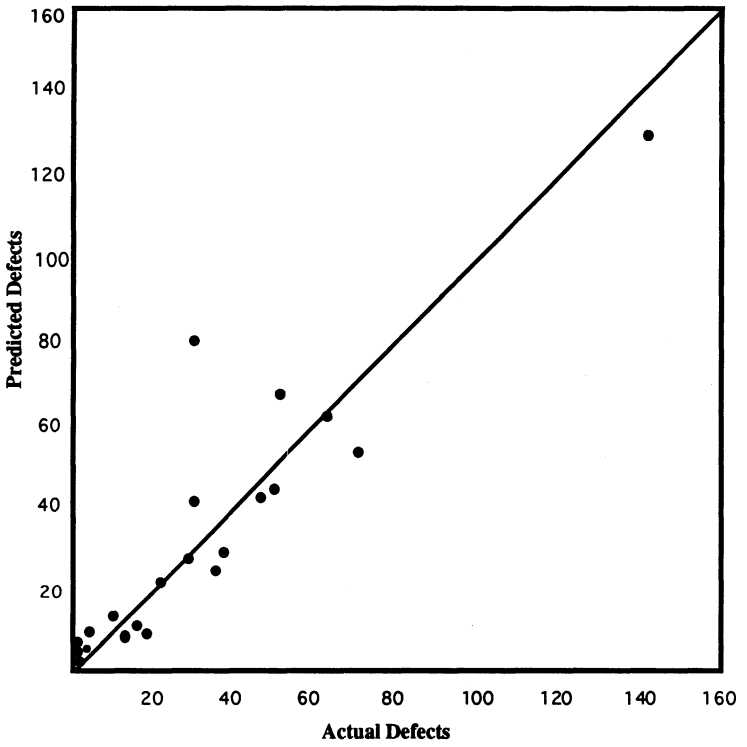

Figure 1 Actual Defects vs. Predicted Defects (subsystem rollups)

Table 3 Project Level Aggregations

Project	Actual Defects	Predicted Defects	Actual Defect Density	Predicted Defect Density
A	425	434	9.4	9.6
B	182	179	10.3	10.2
C	419	398	13.5	12.8
D	12	26	4.3	9.3

5 MODEL APPLICATION

In this section, Model B of Table 2 is applied to a project coded in the C programming language. The purpose of this exercise is to demonstrate the cross-language applicability of the results for Ada subprograms. The project involves 2221 subprograms in fourteen subsystems comprising about 184,000 source lines of C code. The analyzer used to extract the subprogram level software complexity characteristics did not provide a count of the number of parameters, hence necessitating the use of Model B. Also, defect data for this project was available only at the subsystem level, requiring the aggregation of subprogram level predictions to the subsystem level. The reuse was measured in terms of the fraction of code reused within a subsystem denoted by FREUSE.

Because of these dissimilarities, several adaptations of Model B in Table 2 were required. Using the coefficients from Table 2 associated with the software complexity measures, we defined a software complexity term, ICV, given by:

$$\ln(ICV) = .25*\ln(1+NSC) + .27*\ln(CC) + .47*\ln(1+ND) \quad (5)$$

From this equation, the ICV was computed for each of the 2221 subprograms. The ICV's were summed to the subsystem level, yielding subsystem complexity levels denoted by TICV. The correlation between the TICV measure and the numbers of defects at the subsystem level was found to be .83.

Next a Poisson model for the expected number of defects at the subsystem level, λ_s, was defined as a logarithmic function of TICV and FREUSE, and estimated on the basis of the subsystem level defect data yielding:

$$\ln(\lambda_s) = \underset{(.55)}{-2.86} + \underset{(.08)}{.93}*\ln(TICV) - \underset{(.41)}{1.36}*\ln(FREUSE) \quad (6)$$

The coefficient estimates enter with the appropriate signs and are significant to within the 5% level of significance.

Equation (6) was used to calculate the predicted defects at the subsystem level which was then correlated with the actual subsystem defects yielding a coefficient of correlation of .86. The plot of predicted vs. actual defects is shown in Figure 2.

6 CONCLUSIONS

A prime objective of this study was to demonstrate the feasibility of a methodology to integrate metrics and provide defect predictions for small software components such as subprograms. Previous approaches based on discriminant analyses have been capable only of identifying defect-prone components as indicated by some defect number cutoff. The use of Poisson analysis overcomes this limitation.

The analyses have led to some insights for software development policy when employing reuse and for software development taking place in highly volatile environments. Changes in software not anticipated in the original design have a strong impact on defect numbers. These changes may result from the need to adapt previously developed software components for reuse or from volatile software requirements. In any case, software components subjected to such changes should be targeted for additional testing.

The analyses were based on data from projects, some of which reused software components developed in other projects. However, we may consider using the models to predict defects for:

- software development projects involving multiple builds
- software maintenance projects involving enhancements
- software integration efforts

For phased software projects involving builds, defect prediction models can be calibrated on the basis of one or more previous builds. The models can then be applied to predict defects for future builds. Software components from previous builds may be regarded as reused and are classified as being used verbatim, with slight modifications, or with extensive modifications. The software components developed for the new build are regarded as new code. Unanticipated

changes in requirements from build to build resulting in non-defect changes may also be taken into account.

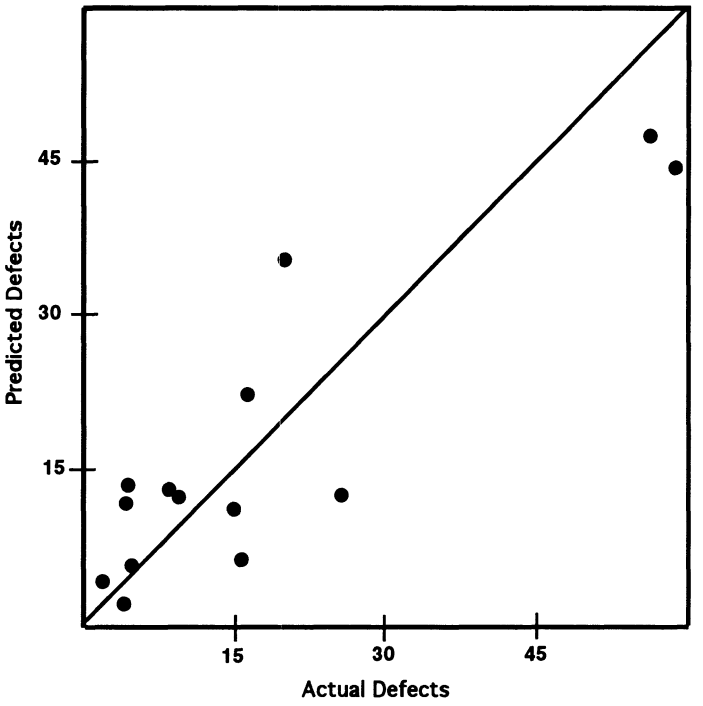

Figure 2 Actual Defects versus Predicted Defects for C Code

For software maintenance projects involving enhancements, the original software components may be regarded as reused. Some of these reused components may require modifications. Any additional software components required to effect the enhancements are regarded as new code. Software defect models may be calibrated on the basis of the original software and then applied to the enhancement project.

Large system development projects often involve the integration of subsystems consisting of COTS software products and developed software subsystems. Their integration may require software component modification at the interfaces while most of the software components "internal" to the COTS subsystems remain unchanged. In addition, some new components may be required to effect the integration. Once again, defect models can be applied to the integration effort and can aid in the development of test strategies.

Future extensions of these models will be concerned with their calibration for other programming languages such as FORTRAN and C to further demonstrate the cross-language applicability of the models. Within the Ada software system context, attention will be focused on the merging of the LVM results with the object-based features of Ada83 and the object-oriented features of Ada95. Also, a continuing validation of these models will be conducted to assess their applicability to a variety of development contexts.

7 REFERENCES

Agresti, W., and W. Evanco (1992), Projecting Software Defects from Analyzing Ada Designs, *IEEE Transactions on Software Engineering*, **18**, 988-997.

Agresti, W., W. Evanco, M. Smith (1990), Early Experiences Building a Software Quality Prediction Model, in *Proceedings of the Fifteenth Annual Software Engineering Workshop*, NASA/GSFC.

Basili, V. R. and Rombach, H. D. (1988) The TAME Project: Towards Improvement-Oriented Software Environments, *IEEE Transactions on Software Engineering*, **6**, 758-773.

Card, D. and W. Agresti (1988), Measuring Software Design Complexity, *Journal of Systems and Software*, **8**, 185-197.

Cote, V., P. Bourque, S. Oligny, and N. Rivard, (1988), Software Metrics: An Overview of Recent Results, *Journal of Systems and Software*, **8**, 121-131.

Doubleday, D. L. (1987), *ASAP: An Ada Static Source Code Analyzer Program*, Technical Report 1895, University of Maryland, College Park, Maryland.

Evanco, W. and W. Agresti (1992), Statistical Representation and Analyses of Software, in *Proceedings of the Seventeenth Symposium on the Interface of Computer Science and Statistics*, College Station, TX., 334-347.

Evanco, W. and R. Lacovara (1994), A Model-Based Framework for the Integration of Software Metrics, *Journal of Systems and Software*, **26**, 77-86.

Goel, A. L. and K. Okumoto (1979), Time Dependent Error Detection Rate Model for Software Reliability and Other Performance Measures, *IEEE Transactions on Reliability*, **28**, 206-211.

Henry, S. and D. Kafura (1981), Software Structure Metrics Based on Information Flow, *IEEE Transactions on Software Engineering*, **7**, 510-518.

Kafura, D. G., and G. R. Reddy, (1987), The Use of Software Complexity Metrics in Software Maintenance, *IEEE Transactions on Software Engineering*, **13**, 335-343.

McCabe, T. J., (1976), A Complexity Measure, *IEEE Transactions on Software Engineering*, **2**, 308-320.

Munson, J. C., and T. M. Khoshgoftaar (1992), The Detection of Fault-Prone Programs, *IEEE Transactions on Software Engineering*, **18**, 423-432.

Musa, J. D. (1993), Operational Profiles in Software Reliability Engineering, *IEEE Software*, **10**, 14-32.

Nielson, K. and K. Shumate (1988), *Designing Large Real-Time Systems with Ada*, McGraw-Hill Book Company, New York, New York.

Selby, R. W. and A. Porter (1988), Learning from Examples: Generation and Evaluation of Decision Trees for Software Resource Analysis, *IEEE Transactions on Software Engineering*, **14**, 1743-1757.

Walsh, T. J. (1979), Software Reliability Study Using a Complexity Measure, in *Proceedings of the National Computer Conference,* New York: AFIPS, 761-768.

Zuse, H. (1990), *Software Complexity: Measures and Methods*, Walter de Gruyer and Company, New York, New York.

8 ACKNOWLEDGEMENTS

The author thanks Mr. Frank McGarry and Mr. Jon Valett of the Software Engineering Laboratory at the NASA Goddard Space Flight Center for their cooperation in providing the data used in this analysis. This research was conducted through funds provided by the MITRE Technology Program.

9 BIOGRAPHY

William M. Evanco received the B.S. degree in physics from Carnegie-Mellon University, Pittsburgh, PA, and the Ph.D. degree in theoretical physics from Cornell University, Ithaca, NY. He has been with the MITRE Corporation since 1987. His primary research interests are software metrics and software performance modeling.