Identifying Error-Prone Software—An Empirical Study

VINCENT Y. SHEN, TZE-JIE YU, STEPHEN M. THEBAUT, MEMBER, IEEE, AND LORRI R. PAULSEN

Abstract—A major portion of the effort expended in developing commercial software today is associated with program testing. Schedule and/or resource constraints frequently require that testing be conducted so as to uncover the greatest number of errors possible in the time allowed. In this paper we describe a study undertaken to assess the potential usefulness of various product- and process-related measures in identifying error-prone software. Our goal was to establish an empirical basis for the efficient utilization of limited testing resources using objective, measurable criteria. Through a detailed analysis of three software products and their error discovery histories, we have found simple metrics related to the amount of data and the structural complexity of programs to be of value for this purpose.

Index Terms—Defect density, error-prone modules, probability of errors, program testing, software errors, software metrics.

I. Introduction

INTEREST in the field of program testing continues to grow—as does the demand for complex, and reliable, programs. In attempting to meet this demand, successful producers of commercial software have learned two important principles.

1) It is extremely important that software be delivered to customers on schedule and with the fewest number of errors possible.

2) The cost of correcting program errors can (and typically does) increase enormously with time to discovery.[1]

Consequently, techniques which facilitate the early detection of the majority of programming errors have been actively sought. One approach is to selectively focus available resources on those components of a software system which are believed to have the highest concentration of defects. The key, of course, is to identify these components as *early* in the development process as possible.

In this paper, we describe a study undertaken to assess the potential usefulness of various product- and process-related measures[2] in identifying those components of large software systems which are most likely to contain errors. Our goal was to establish an empirical basis for the use of objective measurable criteria in developing cost-effective strategies for program testing. The study was based on the detailed analysis of three program products developed at IBM's Santa Teresa Laboratory—a large and modern software production facility. The development approach employed was highly disciplined, and involved two separate phases of program testing—first by the development team, and then by independent test groups during and after code integration. We begin by describing these phases in some detail.

II. Program Testing at Santa Teresa

The first phase of program testing begins early in the development process with a series of design reviews and code inspections. Following these, "unit testing" is performed to verify the correct operation of each module in isolation. Errors detected during this phase are recorded informally and are corrected by the individual programmers responsible. Such errors were not considered in this study as the data were not available.

The second phase consists of a formal series of tests performed by two independent testing organizations. The intent is to ensure that the program satisfies its specified functional requirements. There are two basic phases involved.

1) Development Verification Test (DVT): Conducted at the time of initial system integration, this series of tests is designed to 1) remove defects from any new function added to the product, and 2) verify that all functions provided in the previous version of the program continue to operate as specified. Any defects discovered are formally recorded and reported for corrective action. The support system used to trace the status of each error report, or *program trouble memorandum* (PTM), provided a convenient and objective mechanism for collecting error data.

2) Manufacturing Volume Test (MVT): This series of tests is designed to find and correct problems which affect the ease-of-use, installation, and conversion required to put the new product in service. A separate testing organization performs the test. Any additional deficiencies discovered are recorded as PTM's.

Errors that are discovered and corrected following product release result in the issuance of an *authorized program analysis report* (APAR). For the products considered in our study, complete APAR data were made available.

III. Software Products

The products studied were developed and released since 1980. Product A is a software metrics counting tool written primarily in Pascal, and was designed for the internal use of various de-

Manuscript received June 29, 1984; revised November 5, 1984.

V. Y. Shen and T. J. Yu are with the Department of Computer Sciences, Purdue University, West Lafayette, IN 47907.

S. M. Thebaut is with the Department of Computer and Information Sciences, University of Florida, Gainesville, FL 32611.

L. R. Paulsen is with the Santa Teresa Laboratory, IBM Corp., San Jose, CA 95150.

[1] The experience of one large producer, for example, suggests cost ratio differences of 1:20:80 for corrections made prior to, during, and after formal test, respectively [10].

[2] "Product-related" measures are those which can be derived from some representation of the program itself (e.g., source code, pseudocode, design specifications, etc.). "Process-related" measures are those which can be derived from information related to the process by which a program is developed, tested, modified, etc.

0098-5589/85/0400-0317$01.00 © 1985 IEEE

TABLE I
PRODUCT SUMMARY

Product	Modules	KTSI	KCSI	Language
A	25	7	7	Pascal
B1	253	86	86	PL/S
B2	253	89	3	PL/S
B3	258	94	5	PL/S
C	639	326	60	Assembly

velopment organizations within IBM. Product B is a compiler written primarily in PL/S (system derivative of PL/1), and has been released in three successive versions (B1, B2, and B3). Our study incorporates data from each version. Product C is a database system written primarily in Assembly language.

Programs are composed of separately compilable subprograms called "modules;" typically, each module supports one or more system functions. For our purposes, each module was classified as one of the following types.

1) BASE modules–unchanged modules from a previous release of the same product;

2) MODIFIED modules–changed modules from a previous release of the same product;

3) NEW modules–modules that did not exist in a previous release of the same product;

4) TRANSLATED modules–modules taken from a different product which require language translation but no change in logic.

About 50 percent of the modules (126 out of 253) in the first release of Product B were adapted from another product written in a different language. These modules were translated into PL/S, either manually or with the aid of a language translation tool developed within IBM.

The size and principal language of the products considered are given in Table I. The column labeled "KTSI" shows the total number of source instructions (excluding comments), in thousands. The column labeled "KCSI" shows the total number of NEW, MODIFIED, or (in the case of product B1) TRANSLATED source instructions, in thousands. Thus, KTSI and KCSI entries are the same for the two "newly developed" products, A and B1.

The APAR records available for products A, B1, B2, and C were essentially complete; each had seen wide use and few additional APAR's were expected. Product B3, however, was relatively new and had seen less use than others. Thus, results of our APAR-related analysis for B3 should be weighed accordingly.

IV. THE SOFTWARE METRICS

A program analysis tool in use at Santa Teresa was made available for the study. It provides the basic metrics of Halstead's Software Science [4]:

η_1–the number of unique *operators*;
η_2–the number of unique *operands;*
N_1–the total number of *operators*; and
N_2–the total number of *operands*.

In addition, the tool was extended for this study to provide

DE–the total number of *decisions*

which is closely related to McCabe's graph theoretic measure $v(G)$ [5]. *DE* is simply a count of the conditional statements, loops, and Boolean operators such as AND, OR, NOT, etc.

Also considered was the difference in metric vlaues from one product version to another. For example, the magnitude of the difference in η_1 for two successive versions of a given module is denoted $\Delta\eta_1$. We will refer to these measures as *change* metrics to distinguish them from others.

The basic Software Science metrics were combined by Halstead in a number of ways to produce additional measures. Considered were

$\eta = \eta_1 + \eta_2$ - the *Vocabulary* measure,
$N = N_1 + N_2$ - the *Length* measure,
$V = N \times \log_2 \eta$ - the *Volume* measure,

$D = \frac{\eta_1}{2} \times \frac{N_2}{\eta_2}$ - the *Difficulty* measure, and

$E = D \times V$ - the *Effort* measure.

It has been shown that N and V are closely related to the traditional size measure, lines of code (LOC) [2].

For MODIFIED modules, we also considered a number of *change* and *nonchange* metric combinations (e.g., $\Delta N \times D$) supported in an earlier study with a smaller database [9]. By restricting our interest to a small number of previously supported measures, we were able to conduct a more thorough analysis than would otherwise be possible.

In keeping with our goal to provide tools for identifying error-prone code in a *timely* fashion, we have identified four stages of program development, each of which may be associated with a given set of product- and/or process-related measures. These stages are as follows.

1) Start of Program Design: Even before the completion of program design, a project manager will know something of the general strategy to be employed. Will the program build upon a previous version? Can some modules be taken from another product and translated for this version? To represent these different approaches, Boolean variables were defined and assigned values on a module-by-module basis according to the information available. In this way, the relationship between program reliability and development strategy could be studied statistically.

2) End of Program Design: At this stage, it may be possible to accurately estimate the number of unique operators (η_1), unique operands (η_2), and decisions (*DE*). The accuracy of such estimates will probably depend on the development strategy employed.[3]

3) End of Program Coding–Start of Integration Test: With coding complete, all metrics derivable from source code analysis become available for study.

[3] A recent study at Purdue University, for example, showed that approximately 60 percent of a group of well-disciplined programmers were able to estimate η_2 after program design within 25 percent of their final values in two experiments [15].

4) End of Test: At this point, in addition to the information already available, test results (e.g., number of PTM's) for each module are known.

Most of the metrics considered in this study—in particular, the product-related metrics—are based solely on the automated analysis of program source code using the tool at Santa Teresa. Others, i.e., the process-related metrics, are based on what is often manually recorded information about the course of project development. Therefore, they tend to be sensitive both to errors in recording and to simple changes in policy or practice. One process-related metric, the count of defects discovered during testing, has been used as an independent variable in several defect prediction models [6], [10], [12]. A major objective of this study was to identify product-related metrics which might supplant the use of measures such as this.

V. Defect Prediction Models

For a metric to be useful in deciding how best to allocate limited testing resources, a statistical relationship should exist between its value and the relative error-proneness of modules. In our attempt to identify such metrics, we have focused primarily on statistical models of the *linear* type (i.e., models of the form $\hat{Y} = b_0 + b_1 X_1 + b_2 X_2 + \cdots$). There are powerful tools available for the analysis of these models, and they often serve as satisfactory first approximations to the more complex relationships which may actually exist.

A. Cumulative Errors

The *count* of discovered *module defects* C_{MD} is the dependent variable in this case. This is simply a count of the PTM's and APAR's associated with each program module. A regression-derived estimator for C_{MD} will be written as $\hat{C}_{MD}$. Since, as suggested above, the actual number of reported defects tends to depend on a number of project-dependent factors, $\hat{C}_{MD}$ will be useful chiefly in describing the *relative* error-proneness of modules.

B. Postrelease Errors

Since the nature and significance of individual programming errors can and do vary greatly, it is probably reasonable to compare modules on the basis of reported defects only when the numbers considered are large. Such is the case, for example, when considering C_{MD} since most modules have many PTM's associated with them. With APAR counts alone, however, this is not the case. For the products considered in this study, most modules had no APAR's associated with them, while those that did typically had only one or two. For this reason, a binary dependent variable, P_{APAR} with value *zero* for no associated APAR's, and value *one* otherwise, was used. A regression-derived estimator for P_{APAR} (written $\hat{P}_{APAR}$) has an interesting and potentially useful interpretation: it represents the *expected probability* that a particular module will be associated with one or more APAR's. As with C_{MD}, however, estimators for P_{APAR} should be interpreted carefully. Since the number of errors reported after product release will likely be affected by such factors as program use, testing effort expended, and reporting methods used, generalizations should be limited to those based on the observed *distribution* of reported errors. Generalizations such as "NEW modules are more likely to contain errors than BASE modules" *may* be appropriate while those such as "the probability is 0.28 that NEW modules will contain errors" are almost certainly not.

TABLE II
BASE-NEW-MODIFIED-TRANSLATED MODULE COMPARISONS: C_{MD}

Product	Language	BASE		NEW		MODIFIED		TRANSLATED	
		$\bar{C}_{MD}$	*count*	$\bar{C}_{MD}$	*count*	$\bar{C}_{MD}$	*count*	$\bar{C}_{MD}$	*count*
A	PASCAL			4.40	25				
B1	PL/S			8.30	127			4.47	126
B2	PL/S	0.43	157			5.07	96		
B2(MOD)*	PL/S			5.34	75			4.10	21
B3	PL/S	0.06	123	0.00	5	3.53	130		
C	BAL			9.00**	43	5.98**	334		
C	PL/S			0.40	50				

*These are MODIFIED modules in B2 which were NEW or TRANSLATED in B1.

**Though the difference between 9.00 and 5.98 appears large, the associated level of significance is low ($\alpha > .25$).

VI. Evaluation Criteria

For each model considered, an "all possible regressions" search procedure was used to identify the "best" set of one, two, and three independent variables from the pool of variables available at each stage of program development (Section IV). This approach was made possible by placing a reasonable limit on the total number of variables considered. Standard statistical criteria were employed in the process.[4] One of these, the coefficient of multiple determination (R^2), should be familiar to most readers. It may be interpreted as the proportion of variation in the dependent variable (e.g., C_{MD}) associated with (but *not* necessarily caused by) that in the independent variable(s). Where appropriate, R^2 values will be reported for the models discussed.

VII. Results of Analysis

The findings are organized according to stage of program development. For each stage we identify the variables considered, the relevant data, and the major results.

A. Start of Program Design

Only those variables representing the general development strategy were considered. As defined in Section III, modules were classified as either NEW, MODIFIED, TRANSLATED, or BASE. We are interested in seeing whether different strategies lead to different characteristics of module errors for this stage. Table II shows the average number of cumulative module defects ($\bar{C}_{MD}$) for each category, and Table III shows the average expected probability of postrelease defects ($\bar{P}_{APAR}$). The count of modules observed in each category is also given (*count*). Note that there are no MODIFIED PL/S modules for

[4] See, for example, [7, pp. 376–382].

TABLE III
BASE–NEW–MODIFIED–TRANSLATED MODULE COMPARISONS: $\bar{P}_{APAR}$

Product	Language	BASE		NEW		MODIFIED		TRANSLATED	
		$\bar{P}_{APAR}$	*count*	$\bar{P}_{APAR}$	*count*	$\bar{P}_{APAR}$	*count*	$\bar{P}_{APAR}$	*count*
A	PASCAL			0.32	25				
B1	PL/S			0.40	127			0.07	126
B2	PL/S	0.17	157			0.61	96		
B2(MOD)*	PL/S			0.60	75			0.67	21
B3	PL/S	0.03	123	0.00	5	0.28	130		
C	BAL			0.53	43	0.51	334		
C	PL/S			0.28	50				

*These are MODIFIED modules in B2 which were NEW or TRANSLATED in B1.

TABLE IV
BEST PREDICTORS: END OF PROGRAM DESIGN AND END OF PROGRAM CODING

Product	C_{MD}	P_{APAR}	Subset
A	η_2	η_2	
B1	*DE*	η_2	1
B2	η_2	η_2	2
B3	η_2	*DE*	2
C	*DE*	η_2	3
C	η_1	η_2	4

1. NEW modules, excluding table initialization modules.
2. MODIFIED modules, excluding table initialization modules.
3. NEW assembly language modules.
4. MODIFIED assembly language modules.

product C, no MODIFIED or BASE modules for products A or B1, no NEW modules for product B2, and only five NEW modules for product B3. The principal findings from Tables II and III appear below. All reported differences are based on a 0.005 level of significance,[5] and reflect comparisons between module subsets of a given product.

1) As expected, MODIFIED modules (products B2, B3, and C) and NEW modules (product C) were significantly more error-prone than BASE modules.

2) MODIFIED Assembly language modules (product C) were significantly more error-prone than NEW PL/S modules, but *not* significantly more (nor less) error-prone than NEW Assembly language modules.

3) NEW modules (product B1) were significantly more error-prone than TRANSLATED modules.

The term "error-prone" as used here refers to both cumulative and postrelease discovered errors (C_{MD} and P_{APAR}, respectively). The normal strategy of allocating more testing resources to NEW and MODIFIED modules appears reasonable (item 1). Even so, Table III shows that these modules were still responsible for the majority of postrelease errors.

Our finding that NEW Assembly language modules were neither significantly more nor less error-prone than MODIFIED Assembly language modules (item 2) appears to be consistent with that of a recent study by Basili and Perricone [1], who studied the number of errors for modules written primarily in Fortran. This result suggests that the decision to either modify or rewrite an existing module should not, in general, be based on the assumption that one approach is inherently more error-prone than the other. Other factors, such as time schedule or effort, may be more important regarding this decision.

In view of our finding that NEW modules were significantly more error-prone than their TRANSLATED counterparts (item 3), it becomes interesting to ask if this difference holds when these modules are later modified. To answer this question, we considered those TRANSLATED modules of product B1 which were later modified for product B2. Our finding was that modules that were NEW and later MODIFIED were *not* significantly more nor less error-prone than those that were TRANSLATED and later MODIFIED (row B2(MOD) in Tables II and III).

While the findings for this stage may well be of use in the development of general project strategies, it should be noted that the usefulness of any model based solely on the classifications considered here will be limited. To illustrate this point, consider that for non-BASE modules, only about 30 percent of the variance[6] in C_{MD} could be accounted for using these classifications.

Additional results of our analysis will be given for the more error-prone non-BASE modules only. Our approach was to consider individual subsets of NEW or MODIFIED modules for each of the products studied. By doing so, we were able to reduce the number of independent variables under consideration at any given points, and thus, allow for a more thorough evaluation of those measures of particular interest. We also excluded several "table initialization" modules with abnormally high η_2 and low *DE* values.[7]

B. End of Program Design

As discussed previously, good estimates for η_1, η_2, *DE*, and, in the case of MODIFIED modules, their *change* metric counterparts, may be available for analysis at this stage. However, since no such estimates were made during the development of the products under study, we have made use of the final program measures only. The metrics found to be the best predictors of C_{MD} and P_{APAR} (in terms of R^2) are shown in Table IV. A

[5] Standard nonparametric statistical procedures for testing the significance of differences between two or more independent samples were employed (see, for example, [13]). An intuitive interpretation of the 0.005 level of significance is that a difference as great as that observed should occur purely by chance with a probability no greater than 0.005; thus, we may conclude with a high degree of confidence that a "real" difference actually exists.

[6] $R^2_{B2} = 0.33, R^2_{B3} = 0.25$.

[7] There were 19 such modules, all from product B. Each had a large number of constants (e.g., error messages) and few, if any, conditionals.

discussion of their performance in practical terms will be given later.

The following were the principal findings for this stage.

1) The number of unique module *operands* (i.e., variables and constants), η_2, was the best single predictor, overall.

2) For MODIFIED modules, the non*change* metrics performed better than the *change* metrics when considered individually.

3) Regression models which combined a non*change* metric (such as η_2 or DE) with a *change* metric (such as $\Delta\eta_2$ or ΔDE) generally performed better than models which combined metrics of the same type.

For most subsets, the difference in performance between η_2 and DE was small and, in fact, found not to be significant at the 0.01 level.[8] Moreover, we found these measures were themselves highly correlated. For the Assembly language modules of product C, η_1 accounted for slightly more of the observed variance in both C_{MD} and P_{APAR} than did either η_2 or DE, but again, this difference was found not to be significant.

Our finding that *change* metrics did not perform well individually is probably related to the need on the part of programmers for some level of overall module understanding before modifications can be made. In particular, the actual task of modifying a module is probably simpler, in general, than identifying the possible implications of that modification. Thus, metrics which reflect overall module complexity might be expected to perform better than those which reflect only the change itself.

Finally, our finding regarding the relative performance of combined measures is consistent with an earlier result reported by Paulsen, Fitsos, and Shen [9]. We consider this an area deserving further study in the future.

C. End of Program Coding–Start of Integration Test

At this stage, all metrics derivable from source code analysis become available for study. These include all the product-related metrics described in Section IV. We were surprised to find that the set of best predictors for both C_{MD} and P_{APAR} remained unchanged from the previous stage. Thus, Table IV is used for both. Again, for reasons discussed earlier, our results are based on the final program measures.

Table IV shows that the metric η_2 remains the best overall predictor for both C_{MD} and P_{APAR}, followed closely by DE. This was somewhat surprising in view of earlier work supporting measures such as N, V, and D [3], [8], [14]. It is also rather gratifying since, as discussed earlier, reasonable estimates for η_2 and DE may be available sooner than for others.

Interestingly, our finding does provide support of sorts for a much earlier study by Motley and Brooks [6]. By using regression techniques to analyze the error rates for two large software projects commissioned by the Department of Defense, they discovered that of over 50 independent variables considered, the number of implicitly defined variables and the number of unconditional jumps were the best single predictors of errors. The first of these measures is a subset of η_2, while the second is related to DE.

TABLE V
BEST METRICS AT THE END OF TEST

Product	P_{APAR}	Subset
A	η_2	
B1	η_2	1
B2	η_2	2
B3	No. of PTMs	2
C	No. of PTMs	3
C	η_2	4

1. NEW modules, excluding table initialization modules.
2. MODIFIED modules, excluding table initialization modules.
3. NEW assembly language modules.
4. MODIFIED assembly language modules.

TABLE VI
METRIC PERFORMANCE FOR PRODUCT B1: C_{MD}

Metrics	R^2	*PRED* 25	$\overline{MRE}$
η_2	.74	.43	.47
η_2, DE	.78	.45	.44
N	.72	.45	.48

D. End of Test

The number of PTM's associated with each module is now added to the set of available independent variables. Only P_{APAR} is considered as a dependent variable since the PTM measure accounts for 60-90 percent of the variance observed in C_{MD}. The best predictors for this stage are shown in Table V.

Values of R^2 were higher for η_2 than for the number of PTM's in four of six cases. However, the differences were generally small (amounting to around ten percent or less of the observed variance in P_{APAR}), and were found not to be significant.

E. Additional Measures of Model Performance

We now illustrate the performance of some of the metrics considered by examining statistics related to goodness-of-fit for one of the products studied. The product selected for this purpose, B1, was chosen because of its large size, homogeneity in language and module types, and completeness in error data.[9]

Table VI shows the performance in estimating C_{MD}. The statistics given are 1) R^2, the coefficient of determination; 2) *PRED* 25, the percentage of estimates within 25 percent of their observed value; and 3) $\overline{MRE}$, the mean magnitude of relative error, defined as

$$\overline{MRE} = \frac{1}{n} \sum_{i=1}^{n} \left| \frac{C_{MD} - \hat{C}_{MD}}{C_{MD}} \right|.$$

[8] A test of significance based on Fisher's z' Transformation was used (see [7, pp. 404-407]).

[9] As discussed in Section VII-B, only NEW modules (excluding table initialization modules) were considered in the analysis.

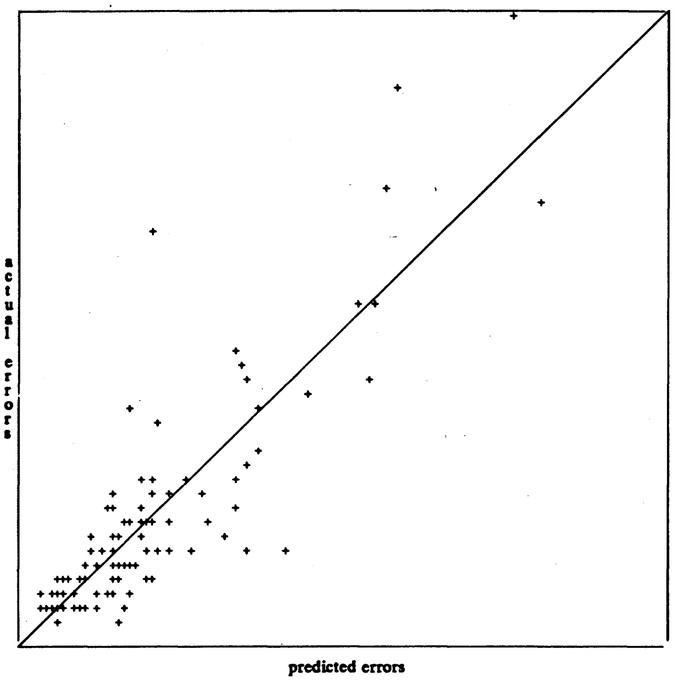


Fig. 1. Cumulative error prediction ($\hat{C}_{MD}$) using η_2 (product B1).

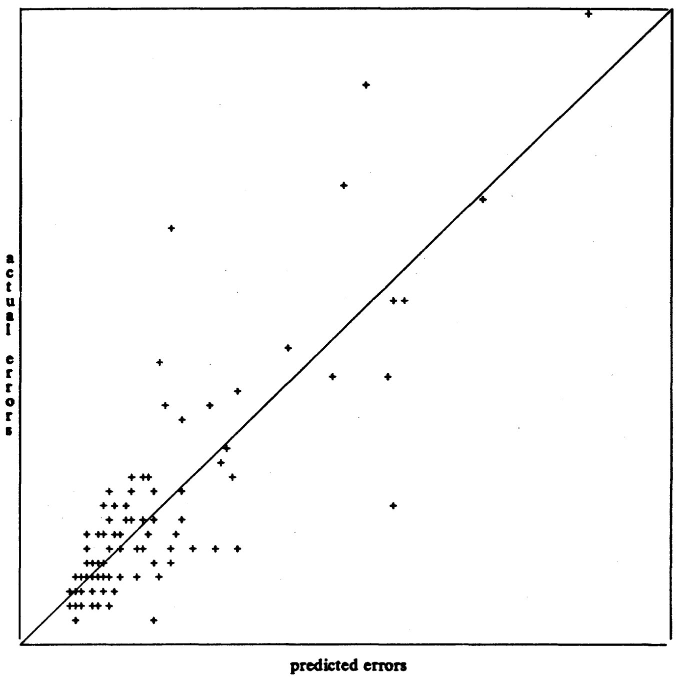


Fig. 3. Cumulative error prediction ($\hat{C}_{MD}$) using N (product B1).

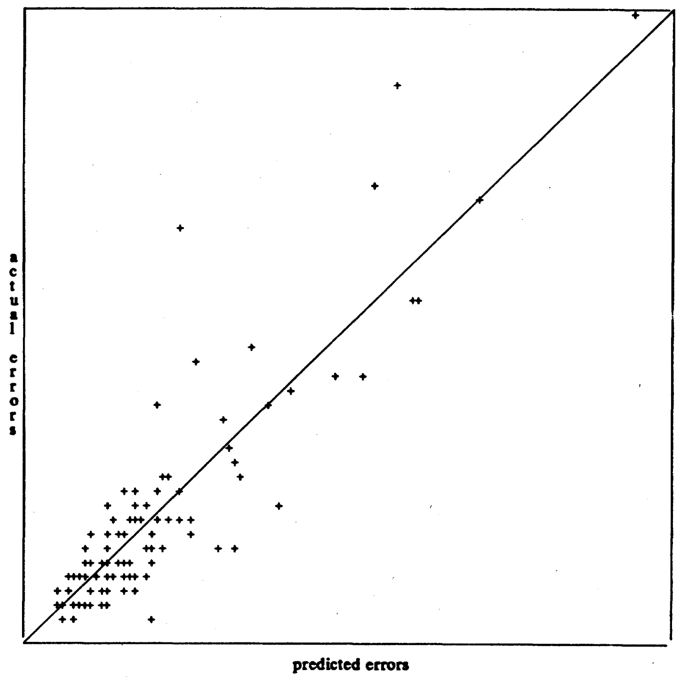


Fig. 2. Cumulative error prediction ($\hat{C}_{MD}$) using η_2 and DE (product B1).

Figs. 1-3 show the scatter plots of actual versus predicted module defects (i.e., C_{MD} versus $\hat{C}_{MD}$) using η_2, η_2 and DE, and N as independent variables.

Consider now a simple scheme for classifying modules into one of two groups according to the likelihood of their being associated with postrelease errors. If $\hat{P}_{APAR} \geqslant 0.5$, classify the module as high risk, otherwise, low risk. Using this scheme, 75 percent of the modules in B1 will be classified correctly using η_2 as the independent variable. Using N as the independent variable, 71 percent will be classified correctly. Finally, if the number of PTM's is used as the independent variable, 72 percent will be classified correctly. Note that these values reflect the goodness-of-fit of these models with the data used to develop them, and are therefore higher than would be expected if the models were applied to new data.

Since the differences in performance for N, η_2, DE, and the number of PTM's was typically small for the products studied, the issue of how soon these metrics can be accurately estimated becomes quite important.

VIII. The Density of Defects

We have used two indexes of module error-proneness: the count of discovered defects (C_{MD}) and the occurrence of one or more postrelease discovered defects (P_{APAR}). Each of the metrics found to be a good predictor of these measures was also found to be related to module size. In general, larger modules simply tend to have more errors than smaller ones. For this reason, "error density"—usually defined as the number of defects per 1000 LOC—has been widely used as a "size-normalized" index of quality (see, for example, [1]). Implicit in this notion is the assumption that error density and module size are unrelated. Our work, however, suggests a somewhat different conclusion. Consider, for example, the plot of C_{MD}/N versus N for product B1 as shown in Fig. 4.

The observable trend, that there is a higher mean error rate in smaller sized modules, is consistent with that discovered by Basili and Perricone [1]. They attributed the trend to the distribution of interface errors, the extra care taken in coding larger modules, and the possibility of undetected errors in larger modules.

Whatever the reasons, our finding that $b_0 + b_1N$ is a better model for C_{MD} than is b_1N, is consistent with this result.

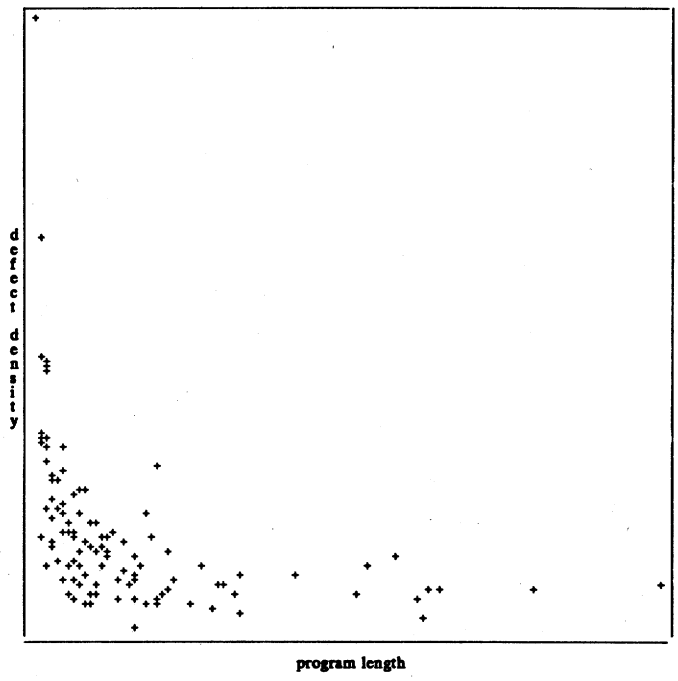


Fig. 4. Defect density (C_{MD}/N) versus module size (N) for product B1.

For if

$$C_{MD} \approx b_0 + b_1 N$$

then error density becomes

$$\frac{C_{MD}}{N} \approx \frac{b_0}{N} + b_1.$$

Thus, as N increases, (C_{MD}/N) decreases asymptotically to b_1, as suggested in Fig. 4. Further analysis of the data showed that the minimum size beyond which error density could reasonably be considered unrelated to module size was $N=$ 2500, or approximately 500 lines of code. Only 24 (out of 108) modules were this large or larger. Analysis of products B2 and C produced similar results.

We conclude, therefore, that error density is generally a poor size-normalized index of program quality.

IX. Conclusions

Based on our analysis of three program products and their error histories, we have discovered that simple metrics related to the amount of data (η_2) and the structural complexity (DE) of programs may be useful in identifying *at an early stage* those modules most likely to contain errors. This finding provides an empirical basis for the use of these measures in targeting certain modules for early or additional testing, in order to increase the efficiency of the defect removal process. Alternative measures such as module size (e.g., N) or initial test results (e.g., number of PTM's) have been shown to offer little potential for improving the strategy. Our findings also suggest that it may be beneficial to promote programming practices related to modularization that discourage the development of either extremely large or extremely small modules.

Our study of error density shows that this measure is, in general, a poor size-normalized index of program quality. Its use in comparing the quality of either programs or programmers without regard to related factors such as complexity and size is ill-advised.

Acknowledgment

The authors wish to thank T. Franciotti, P. Hutchings, and H. Remus of IBM for their support of this study. Thanks also to K. Christensen and G. Fitsos, also of IBM, for their contributions during the initial phase of the project. B. Dunsmore of Purdue University made valuable suggestions during the final phase.

Finally, we are indebted to several members of the development and testing teams for products A, B, and C, without whose cooperation and concern for quality this research would not have been possible.

References

[1] V. R. Basili and B. T. Perricone, "Software errors and complexity: An empirical investigation," *Commun. ACM*, vol. 27, no. 1, pp. 42–52, Jan. 1984.

[2] K. Christensen, G. P. Fitsos, and C. P. Smith, "A perspective on software science," *IBM Syst. J.*, vol. 20, no. 4, pp. 372–387, 1981.

[3] A. R. Feuer and E. B. Fowlkes, "Some results from an empirical study of computer software," in *Proc. 4th Int. Conf. Software Eng.*, 1979, pp. 351–355.

[4] M. H. Halstead, *Elements of Software Science*. New York: Elsevier North-Holland, 1977.

[5] T. J. McCabe, "A complexity measure," *IEEE Trans. Software Eng.*, vol. SE-2, pp. 308–320, Dec. 1976.

[6] R. W. Motley and W. D. Brooks, "Statistical prediction of programming errors," Rome Air Devel. Cen., Griffiss AFB, NY, RADC-TR-77-175, May 1977.

[7] J. Neter and W. Wasserman, *Applied Linear Statistical Models* Homewood, IL: Irwin, 1974.

[8] L. M. Ottenstein, "Quantitative estimates of debugging requirements," *IEEE Trans. Software Eng.*, vol. SE-5, pp. 504–514, Sept. 1979.

[9] L. R. Paulsen, G. P. Fitsos, and V. Y. Shen, "A metric for the identification of error-prone software modules," IBM Santa Teresa Lab., San Jose, CA, Tech. Rep. TR-03.228, June, 1983.

[10] H. Remus, "Planning and measuring program implementation," in *Proc. Symp. Software Eng. Environments*, Lahnstein, Germany (Gesellschaft fuer Mathematick und Datenverarbeitung). Amsterdam, The Netherlands: North Holland, 1980, pp. 267–279.

[11] V. Y. Shen, S. D. Conte, and H. E. Dunsmore, "Software science revisited: A critical analysis of the theory and its empirical support," *IEEE Trans. Software Eng.*, vol. SE-9, pp. 155–165, Mar. 1983.

[12] M. L. Shooman, *Software Engineering*. New York: McGraw-Hill, 1983.

[13] S. Siegel, *Nonparametric Statistics for the Behavioral Sciences*. New York: McGraw-Hill, 1956.

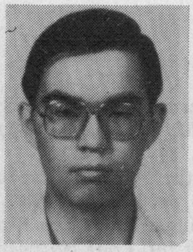
[14] C. P. Smith, "Practical applications of software science—The detection of error prone code," IBM Santa Teresa Lab., San Jose, CA, Tech. Rep. TR03.184, Feb. 1982.

[15] A. S. Wang, "The estimation of software size and effort: An approach based on the evolution of software metrics," Ph.D. dissertation, Dep. Comput. Sci., Purdue Univ., W. Lafayette, IN, Aug. 1984.

Vincent Y. Shen received the Ph.D. degree in electrical engineering from Princeton University, Princeton, NJ, in 1969.

He joined Purdue University, W. Lafayette, IN, in February 1969, and is currently an Associate Professor of Computer Science. He was a Visiting Professor at National Tsing Hua University, Taiwan, Republic of China, during the 1975–1976 academic year. He was a Visiting Faculty

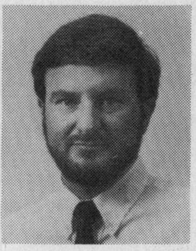
Member of IBM's Santa Teresa Laboratory during the summers of 1982 and 1983. He has published papers in switching and automata theory, computer graphics, operating systems, database, and most recently, software engineering and software metrics.



Tze-Jie Yu received the B.S. degree in electrical engineering from National Taiwan University in 1979, and the M.S. degree in computer science from Purdue University, West Lafayette, IN, in 1983.

From 1981 to the present, he has been a Ph.D. candidate and Research Assistant at Purdue University. He worked for IBM's Santa Teresa Laboratory as a Systems Analyst during the summer of 1983. His research interests include software quality assessment, software tools, and software cost modeling.

Mr. Yu is a member of the Association for Computing Machinery and its Special Interest Group on Software Engineering, and a member of the IEEE Computer Society.



Stephen M. Thebaut (S'79–M'83) received the B.A. degree in mathematics from Duke University, Durham, NC, and the M.S. and Ph.D. degrees in computer science from Purdue University, West Lafayette, IN, in 1979 and 1983, respectively.

He joined the Department of Computer and Information Sciences, University of Florida, Gainesville, FL, as an Assistant Professor in 1983, and was supported by an IBM Postdoctoral Research Fellowship during the 1983–1984 academic year. His current research interests include software reliability, program maintenance, and the large-scale software development process.

Prof. Thebaut is a member of the Association for Computing Machinery, the IEEE Computer Society, and the 356 Registry.

Lorri R. Paulsen received the B.S. degree in mathematics from Wagner College, Staten Island, NY, in 1968.

She joined IBM in 1968, and is currently a Program Development Manager for IBM General Products Division at Santa Teresa Laboratory, San Jose, CA. During her IBM career, she has been associated with systems design and development, working on the ASP Version 3 and JES3 products. In 1979 she became involved with the measurement of programmer's productivity, which led to her research with Software Science metrics and the prediction of error-prone modules. She is currently managing the development of software engineering tools to assist with the design and implementation of IBM products.

Translating SQL Into Relational Algebra: Optimization, Semantics, and Equivalence of SQL Queries

STEFANO CERI AND GEORG GOTTLOB

Abstract—In this paper, we present a translator from a relevant subset of SQL into relational algebra. The translation is syntax-directed, with translation rules associated with grammar productions; each production corresponds to a particular type of SQL subquery.

The translation is performed in two steps, associated with two different grammars of SQL. The first step, driven by the larger grammar, transforms SQL queries into equivalent SQL queries that can be accepted by a restricted grammar. The second step transforms SQL queries accepted by the restricted grammar into expressions of relational algebra. This approach allows performing the second step, which is the most difficult one, on a restricted number of productions.

The translator can be used in conjunction with an optimizer which operates on expressions of relational algebra, thus taking advantage of a body of knowledge on the optimization of algebraic expressions. Moreover, the proposed approach indicates a methodology for the correct specification and fast implementation of new relational query languages. Finally, the translator defines the semantics of the SQL language, and can be used for the proof of equivalence of SQL queries which are syntactically different.

Index Terms—Program translation, query equivalence, query languages, query optimization, relational algebra, relational database model, SQL.

I. Introduction

THE use of the relational model and languages is becoming more and more popular for the development of new database management systems. Formal languages, such as the relational calculus and algebra, have been proposed by Codd [1]

Manuscript received June 13, 1984; revised September 6, 1984. This work was supported in part by a grant from Data Base Informatica.

The authors are with the Dipartimento di Elettronica, Politecnico di Milano, I 20133 Milano, Italy.

0098-5589/85/0400-0324$01.00 © 1985 IEEE