Comparison of Size and Complexity Metrics as Predictors of the Number of Software Faults

Plínio R. S. Vilela[1], Waldo Luis de Lucca[1], Ariane Corso[2], and Mario Jino[3]

[1] Methodist University of Piracicaba, Piracicaba SP, Brazil,
{prvilela,wllucca}@unimep.br
[2] Caterpillar, Piracicaba SP, Brazil.
arianecorso@hotmail.com
[3] State University of Campinas, Campinas SP, Brazil,
jino@dca.fee.unicamp.br

Abstract. Empirical work supporting the hypothesis that simple size metrics and complexity metrics are good predictors of fault-prone modules have been published in the past. Some studies have also shown that contrary to common belief complexity measures are not always better predictors than simple size metrics. All of these studies share one characteristic – they measure size, complexity and number of faults at the granularity of at least modules. In this work we compare the relative ability to predict the number of faults and fault-proneness of simple size metrics, such as Lines of Code (LOC), and complexity metrics. The caveat of our empirical investigation is that we seek evidences to compare the metrics on a level of granularity smaller than modules; we do it for segments of code that implement a particular software testing requirement. Our data show that complexity measures are better predictors of the number of faults than simple size metrics at that granularity level.
Keywords: Size and Complexity Metric Comparison, Software Testing, Faults Predictor, Fault-Proneness.

1 Introduction

Predicting the number of faults hidden in a software is a difficult problem. Nevertheless, research results in this area are very appealing to industry, since information about undiscovered faults are used to support part of software development decision making. A *fault* or likewise a *defect* is commonly defined as a single item in a software artifact that is incorrect [4]; when executed it can generate an inconsistent execution state, an *error*, that may propagate to outside of the system resulting in software behavior that deviates from the software specification – a *failure*.

The relationship between faults and failures in deployed software is not completely understood. It is a fact that pre-deployment undiscovered faults affect overall software reliability; but other factors such as operational profile, the "size" of the fault and fault

[4] A *software artifact* is either a design document, a data structure definition or even source code. We are assuming that faults in earlier stages of software development, if not discovered and removed, eventually make their way into incorrect code.

density also play a part [3]. Collected evidence indicates that a small subset of the software faults is responsible for most of the operational failures [1].

Research efforts in this area have concentrated on the following directions [9]: i) predicting the number of faults of the system; ii) estimating the reliability of the system; and iii) understanding the impact of design and testing processes on fault counts and failure densities. Many researchers have attempted to put forth prediction models. Size and complexity measures are used to predict the number of faults revealed during software testing and operation. Reliability models considering operational profile, testing coverage and software metrics have been proposed to predict the failure rate. We do not attempt to discuss all of these research paths; we concentrate on a comparison of the relative ability of size and complexity metrics in terms of number of faults estimation. For a discussion of most of these research endeavors see Fenton and Neil [9].

We intend to investigate the relative strength of simple size metrics and complexity metrics as the number of faults predictors. The hypothesis is that complexity metrics are better predictors of the number of faults than simple size metrics. The basis for this hypothesis is that the complexity of the software apparently has a more direct influence on the mental burden to the software developer, which in turn leads to a greater number of mistakes and, consequently, more faults. To test this hypothesis we planned and conducted an experiment. Most empirical work on prediction models perform data collection and analysis at the level of at least modules [2, 12, 14, 11, 3, 16, 8]. In our empirical investigation we seek evidences to support the hypothesis at a level of granularity smaller than modules; we do it for segments of code that satisfy a specific property – they implement a particular software testing requirement. This approach has two advantages: i) it reduces threats to external validity of the investigation since small segments of code used in our experiment are representative of small segments of code present in other programs; ii) one application of the types of metrics studied is to help estimate testing effort, segments of code have a direct relationship to testing effort since the tester will have to come up with test cases that cover those testing requirements;

In Section 2 we describe the experiment, defining the method to collect the data and present each of the complexity and simple size metrics used. Section 3 contains the results of our experiment including the data analysis and graphics. In Section 4 we present a discussion of the results presented in Section 3. Finally, in Section 5 we present our conclusions and future work remarks.

2 Description of The Experiment

One fundamental research question is whether software size and complexity are related to the number of faults, specially whether the number of faults is a direct consequence of software size and complexity. This issue is somewhat controvert, specially because even if there is a correlation it is difficult to adequately measure it. One of the hard to prove effects is the tendency to pay more attention to something that is indeed more complex. Human beings – programmers – are usually more careful when dealing with something complex. A complex algorithm will be more carefully implemented, a complex data structure will be more carefully documented and used with greater care. Thus inherent structural complexity affects cognitive complexity, which in turn influences the individual's attitude leading to the greater-care behavior in some individuals and, consequently, smaller number of faults. An individual response to the structural complexity is a factor of his/her own personality and technical knowledge. The purpose of our research is neither to pursue a proof for that effect nor to establish a relationship between metrics and number of faults. Our intention is to compare the metrics among themselves, and the experiment was crafted for that purpose.

2.1 Experiment Design

We first selected a few complexity and size metrics and measured each metric for a number of segments of code from a single program. To evaluate the comparative behavior of each metric with respect to the occurrence of faults we used a program with a controlled number of faults of selected types. A more complex program would not suit our specific needs in this experiment, since the number of documented faults would usually not correspond to the actual number of faults in the program. We needed a program with all faults already known. The selected program was `Cal`, a UNIX program that prints a calendar for a specified year or month. This program has been thoroughly used and debugged; thus, it is reasonable to consider it as correct.

`Cal` has been used in another experiment conducted by Wong, et al. [20, 19]. The preparation for the reported experiment included a systematic injection of a number of faults in it. That setup gave us the appropriate ground to conduct our investigation. The methodology used by the authors to inject faults was intended to guarantee independence from any method to reveal the presence of such faults and to make sure common types of faults were present. A graduate student with at least three years of `C` programming experience was instructed to use experience and judgment to inject one or more faults of each types: *missing path*, *incorrect predicate*, *incorrect computation statement*, *missing computation statement*, *incorrect number of loop iterations*, *missing clause in predicate*; reflecting the types of faults from software faults taxonomies [4, 7, 13].

That procedure resulted in twenty injected faults. We then associate each segment of the code with the corresponding number of faults it contained. Each segment of code represents the part of code that executes a particular def-use association [18]. With that data we are able to calculate the correlation coefficient of metric to number of faults for each of the selected metrics. The correlation coefficient is subsequently used to compare the metrics and establish a partial ordering.

2.2 Subjects

The subjects of our investigation are segments of code that implement def-use associations. Other studies worked on modules to both compute metrics values and count number of faults or measure fault density. Our intention was to repeat this kind of investigation at a level of granularity that is closer to the unit testing activity. During tool supported unit testing a tester runs its unit through a software testing tool, which analyzes the code and derives a list of required elements that should be covered. The nature of the required elements vary with the selected testing criterion; in data flow based testing it is in general a data flow association. Rapps and Weyuker [18] defined a family of such data flow criteria; one of them is the *All-Uses* criterion, which requires exercizing every association between a variable definition and a use in a program.

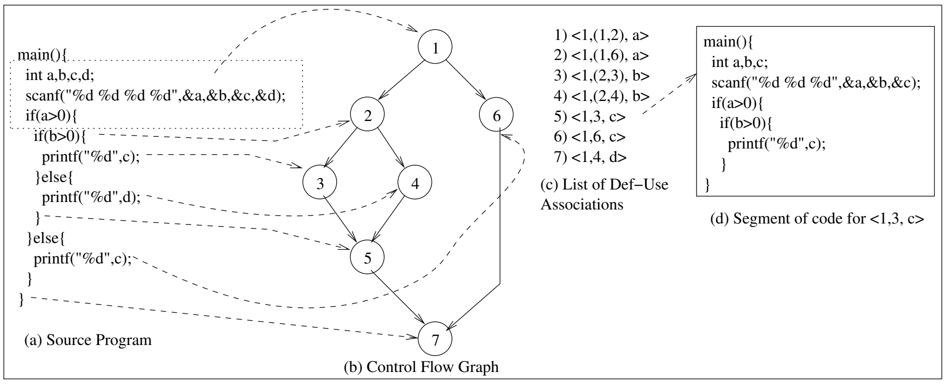


Fig. 1. Def-Use Associations Example

To be able to measure the size and complexity metrics for all the associations in our selected program (`Cal.c`) we had to isolate each segment of code that implemented each of the 215 def-use associations in the program. Figure 1 illustrates this process. The software testing tool used constructs the control flow graph (Figure 1(b)) from the program source code (Figure 1(a)); then it incorporates data flow information to it and computes the list of required def-use associations (Figure 1(c)). We manually extracted from the original source code each segment of the code whose execution covers the corresponding association, Figure 1(d) exemplifies the process for one of the associations in the example. That gives us a total of 215 programs representing the segments of code that implement the associations in `Cal.c`.

The metrics considered in this work are: Lines of Code (LOC, also called Source Lines of Code – SLOC), Halstead's Theory of Software Science metrics [12], and McCabe's Cyclomatic Complexity [15].

To compute *LOC* we counted every line of the program that was not blank or completely included in a comment. Everything else was counted, multiple lines were counted individually even if they were actually part of a single statement.

Halstead's Software Science comprehends several metrics, all based on measures that can be automatically computed from a program, basically the number of operands and operators. Halstead's initial motivation was to prove the hypothesis that the number of operands and operators is strongly correlated to the number of faults discovered in a program. Initial tests of the theory have shown a very high correlation between the software science metrics and measures such as the number of faults in a program [10]. The operands of a program are the variables and constants, while operators (arithmetic operators, boolean operators, keywords and delimiters) affect the value or ordering of operands; both are easily tabulated by a compiler, for example. Four basic measures can be determined from those tabulations: n_1: Number of *distinct operators*, n_2: Number of *distinct operands*, N_1: Number of *occurrences* of *operators*, and N_2: Number of *occurrences* of *operands*.

The size of the *Vocabulary* is $l = n_1 + n_2$, and the *Length* is $L = N_1 + N_2$. Vocabulary and Length are based exclusively on operators and operands counts. We considered these two metrics plus LOC as the set of simple size metrics we use in our experiment. The next Software Science metric is *Volume*: $V = L(log_2 l)$, the intuition underlying this metric is that for each of the L elements in a program $log_2 l$ bits must be specified to choose one of the operators or operands for that element. Thus V measures the total number of decisions a programmer has to make in order to choose the operators and operands in the program. *Program Difficulty* $D = (\frac{n_1}{2})(\frac{N_2}{n_2})$ is directly associated to Volume, but is inversely proportional to the level of abstraction. The Level of Abstraction, $L1$, increases with the number of distinct operands (n_2) and decreases with the number of distinct operators (n_1) and total number of operands (N_2); thus, $L1 = \frac{1}{D}$. *Effort* is the metric that captures the idea that the difficulty of programming increases as the volume of the program increases, and decreases as the program level of abstraction increases, $E = \frac{V}{L1}$.

McCabe's Cyclomatic Complexity may be used to determine the structural complexity of a coded module. This measure is designed to establish a limit to the complexity of a module, intending to improve its readability and reduce its fault-proneness. The primitives are: N, number of nodes (groups of sequential statements that are always executed together); E, number of edges (flow of control between nodes); SN number of decision nodes (nodes with more than one exiting edge); and RG, number of regions (areas bounded by edges with no edges crossings). The complexity of a particular program graph, representing a module, is given by any of the three formulas: $V(G) = RG$, $V(G) = E - N + 2$ or $V(G) = SN + 1$.

We consider Volume, Difficulty, Effort and Cyclomatic Complexity as complexity metrics since they capture some sense of the programmers mental burden to cope with a particular part of the code.

2.3 Method

The four steps used to collect the data are described below.

1. Generate the list of required elements.
   To generate the list of required elements we utilized the tool called Poke-Tool [5] that implements the All-Uses Criterion [17, 18]. The total number of required elements (*def-use associations*) is 215. The program has 4 modules: `main` with 77 def-use associations, `pstr` with 39, `cal` with 88 and `jan1` with 11.
2. For each required element:
   (a) Isolate the corresponding part of the source code.
   To compute the size and complexity at the granularity level of required elements, we had to isolate the part of the code that actually implemented each particular required element. Since a def-use association comprehends the code between a point where a variable is defined and the subsequent point where that variable is used (without an intervening redefinition), we had to separate that part of the code from the rest, and add preceding code to declare and define anything needed to make the new program compile with no problems.

   (b) Compute LOC.
   (c) Count the Halstead's science of software primitives.
   (d) Extract the control flow graph, count arcs and nodes to compute the cyclomatic complexity.
   (e) Count the number of faults.
   The count of faults was made from the initial count of 20 injected faults in `Cal.c`. Each fault counted once for each program that included the segment of code with the fault; thus, a fault could potentially count for a number of programs.
3. Compute the Relative Complexity Index (RCI) for each required element, for each metric.
   This was calculated to normalize the metric values. $RCI_m = (V_m - Min_m)/(Max_m - Min_m)$, where V_m is the calculated metric, Max_m and Min_m are the maximum and minimum observed values for a particular metric.
4. Order the required elements by RCI.

3 Results

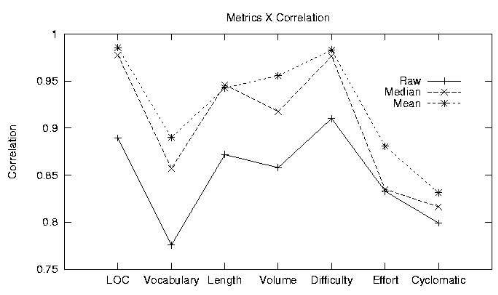
From the collected data we calculated the linear dependency between each metric and the number of faults. The coefficient of correlation is given by Equation 1, where n is the number of points in the sample, each point is a pair (X,Y), where X is the metric value and Y the number of faults, $\overline{x}_{obs}$ and $\overline{y}_{obs}$ are the averages of the observed values for X and Y.

$$\rho_{x,y} = \frac{\sum_{i=1}^{n}(x_i - \overline{x}_{obs})(y_i - \overline{y}_{obs})}{\sqrt{[\sum_{j=1}^{n}(x_j - \overline{x}_{obs})^2][\sum_{j=1}^{n}(y_i - \overline{y}_{obs})^2]}} \tag{1}$$

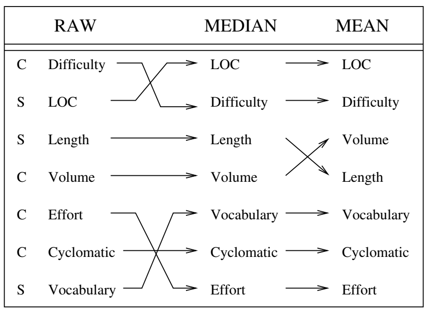
Table 1. Correlation: Metrics x Faults

Metric	Raw	Median	Mean
LOC	0.8895	0.9776	0.9855
Vocabulary	0.7756	0.8571	0.8900
Length	0.8718	0.9459	0.9426
Volume	0.8579	0.9175	0.9556
Difficulty	0.9101	0.9764	0.9830
Effort	0.8329	0.8350	0.8806
Cyclomatic	0.7990	0.9585	0.9786

Table 1 lists all the correlation coefficients calculated for each metric. The first column is the name of the metric, the second column is the correlation calculated from



(a) Filtered Metrics versus Correlation



(b) Metrics Ordered by Correlation

Fig. 2. Metrics *versus* Correlation

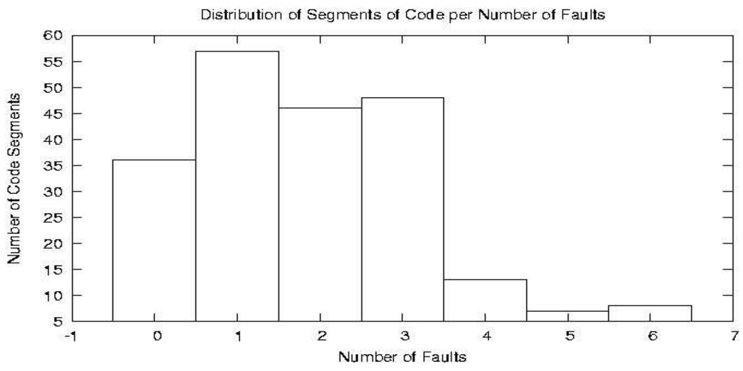
the *RAW* data, including every data point in the sample. In columns three and four we present the correlation coefficients calculated from filtering the number of faults for each metric interval by the *MEDIAN* and *MEAN*, respectively. This filtering was done to measure the ability of a particular metric as a software fault predictor. Figure 2(b) graphically represents the change in the order of better correlation coefficient to worst correlation coefficient. The changes are not very drastic, LOC goes from second to first and Program Difficult goes to second. It is interesting to notice three well defined subsets – the top performers are Program Difficulty and LOC, followed by Length and Volume. The worst performers are Effort, Cyclomatic and Vocabulary. Figure 2(a) depicts the performance of each metric in the three data sets.

The next data analysis was the calculation of the distribution of metric values by number of faults in the data set. Figure 3(a) shows the number of segments of code for each number of faults; for example, there are 57 segments of code with 1 fault, and 48 with 3 faults. Selecting, for example, the 57 segments of code with 1 fault and calculating the Program Difficult metric for each of them we collected a set of measures with the distribution represented in Figure 3(b); the distribution is approximately normal with Mean 14.36 and Standard Deviation 4.15. Figure 3(c) shows the distribution of LOC corresponding to a normal distribution with Mean 11.44 and Standard Deviation 3.83. Both graphics are mean centered and with a unit as a standard deviation.

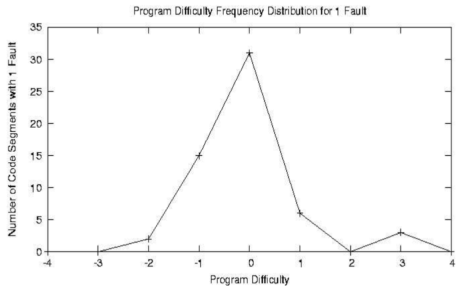
4 Discussion

Analyzing the correlation coefficients presented by the calculations with the whole set of points (*RAW* data column in Table 1), we can see that the complexity metric Program Difficulty presented the highest correlation coefficient, around 0.91, closely followed by the simple size metrics LOC and Length.

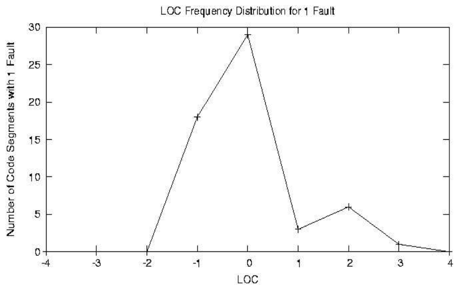
For the median and mean number of faults in each interval of the metric (columns 2 and 3 of Table 1) we see that LOC and Difficulty have their positions inverted; LOC has a better correlation coefficient to the number of faults. Analyzing Figures 4(a) and 4(b) we see that LOC presents dispersion of data points greater than Difficulty when the



(a) Distribution of Segments of Code per Number of Faults



(b) Program Difficulty Frequency Distribution for 1 Fault



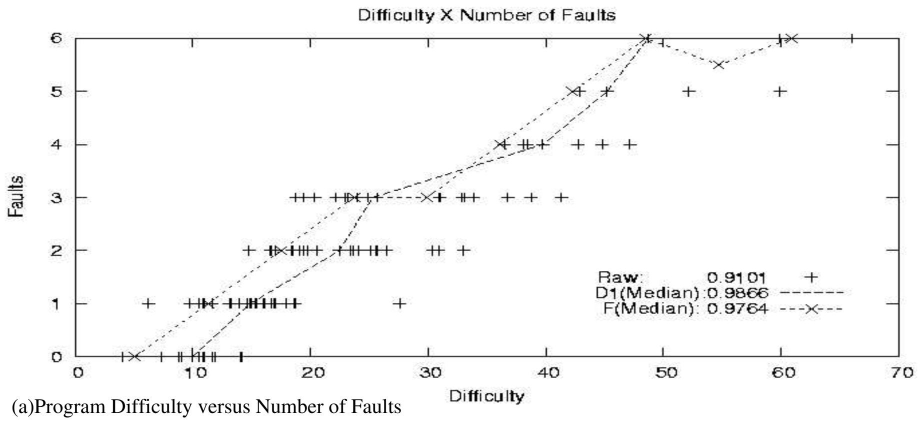
(c) LOC Frequency Distribution for 1 Fault

Fig. 3. Distributions

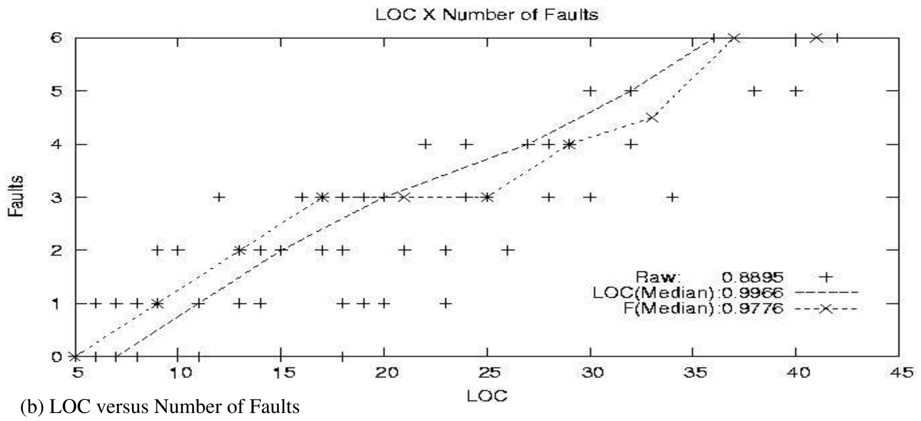
RAW data is considered; when the median is considered the effect of the dispersion is eliminated leading to better results in terms of correlation. Figure 4 also shows the line connecting the points of median value of the metric and its corresponding correlation coefficient.

Even though the median and mean are important tools for data analysis and useful when using the metric values to predict the number of faults in a segment of code, we considered that the analysis of the *RAW* data is more meaningful concerning the real performance of the metrics.

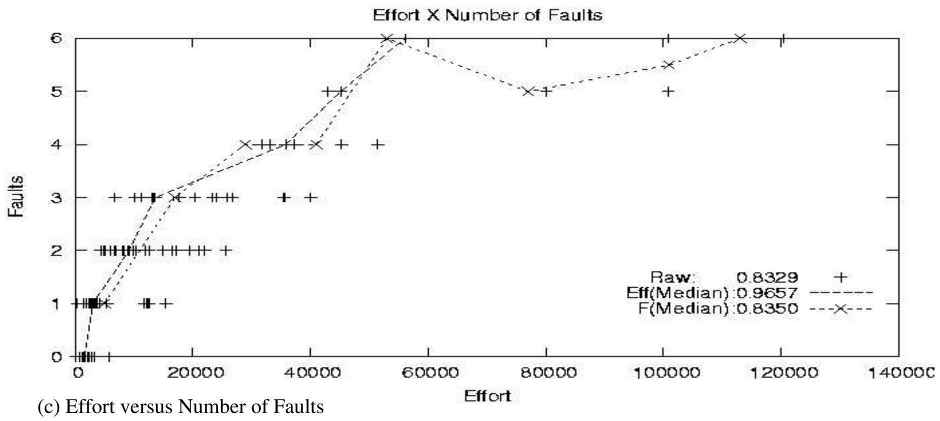
One important analysis to support this discussion is the distribution of segments of code with same number of faults by the corresponding metric measured. For example, Figure 3(b) shows that segments of code with one fault formed a normal distribution of program difficulty metric. That graphic tells us that most of the segments of code with one fault have the program difficulty metric close to a certain mean value. LOC presented similar albeit worse results for this type of analysis, as illustrated by Figure 3(c).



(a)Program Difficulty versus Number of Faults

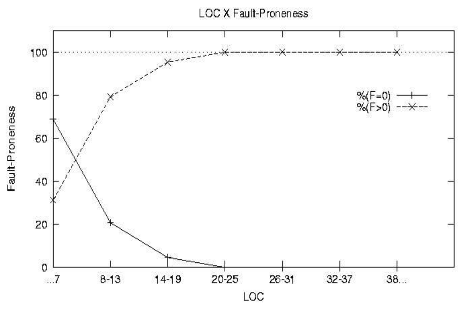


(b) LOC versus Number of Faults

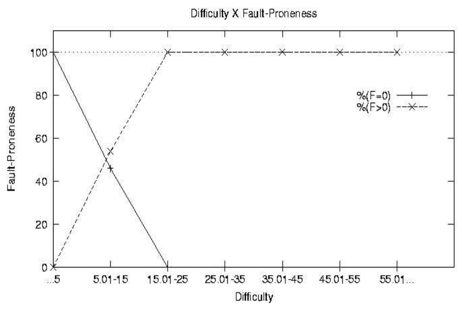


(c) Effort versus Number of Faults

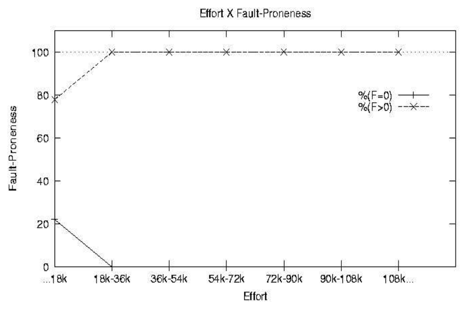
Fig. 4. Metrics *versus* Number of Faults



(a) LOC versus Fault−Proneness



(b) Difficulty versus Fault−Proneness



(c) Effort versus Fault−Proneness

Fig. 5. Metrics *versus* Fault-proneness

Another interesting characteristic to analyze is the fault-proneness. *Fault-proneness* of a part of a software is defined as the probability that it contains at least one fault [5].

For that kind of analysis LOC and Difficulty presented similar performance. Figure 5(a) indicates that the probability of at least 1 fault occurrence gradually increases with LOC, reaching 100% around 20 to 25 LOC. Difficulty, Figure 5(b), also has a gradual increase, reaching 100% after the second metric interval.

4.1 Internal Validity

We identify some threats to internal validity, one is the experimenter bias during the fault injection procedure. To avoid that bias we used a program with faults independently injected in a different experiment, conducted by a different research group.

Another is the distribution of faults. In a real program with real faults we cannot control the distribution of faults or the fault types. To avoid this problem we decided to use a program with injected faults. The fault injection was done by a student, intructed

[5] The definition of fault-proneness is traditionally associated to modules; we generalized it to incorporate the fault-proneness of any segment, or part of the software, even if it is not a complete module.

to use his experience, common sense and judment to realisticly inject a number of faults in the program from a list of common types of faults collected from the literature.

4.2 External Validity

Our concerns with external validity is centered around how representative our experiment subjects are. It is always discussed whether to use real programs of faked programs in software engineering experiments. In the level of granularity we are operating the assumption is that segments of code in our program are representative of segments of code in other programs regardless of program size.

5 Conclusions

Predicting the number of hidden faults in a software is a difficult problem; research results in this area are very appealing to industry, since information about undiscovered faults are used to support a number of software development decision, specially during software testing.

In this work we seek empirical evidences for a hypothesis related to the use of size and complexity measures as software fault predictors: Complexity metrics are better predictors than size metrics. The special characteristic of our investigation is that we seek evidences to support this hypothesis at a level of granularity smaller than modules; we do it for segments of code that implement a data flow software testing requirement

In our experiment we selected the program `Cal.c`. It displays a calendar on the console and accepts a few number of parameters that control which month and/or year it should display plus a few other thing; it runs under `UNIX`. In addition to that, `Cal.c` has a number of documented software faults. We selected a few complexity and size metrics and measure each metric for a number of segments of code of `Cal.c`. We associated each segment of the code with the correspondent number of faults it contained. With that data we calculated the correlation coefficient of metric to number of faults for each of the selected metric.

Our results provide evidences that the hypothesis hold. Some interesting results were also reached. LOC did perform very well; it was almost as good as Program Difficulty. Effort and Cyclomatic, both complexity measures, did not do so well. Effort, for example, presented poor results in terms of correlation, Figure 4(c), and in terms of fault-proneness indicator, Figure 5(c).

As future research and development endeavors we intend to investigate the correlation of Mutation Analysis [6] and number of faults, by calculating the correlation coefficient for the number of mutants generated for a particular segment of code and the number of faults present in the segment. This is to try to come up with empirical evidences to support the hypotheses that the number of mutants is a good estimator of the number of faults and a good indicator of fault-proneness. We also intend to work on automating the manual steps of our data collection methodology to allow some of our ideas to form the basis for the definition of new software testing criteria based on information about requirements complexity and size.

Acknowledgments

The authors would like to thank Prof. Luis A. Consularo for the very helpful and interesting discussions and suggestions about this work.

References

1. E. Adams. Optimizing Preventive Service of Software Products. *IBM Research Journal*, 28(1):2–14, 1984.
2. F. Akiyama. An Example of Software System Debugging. *Information Processing Letters*, 71:353–359, 1971.
3. V. Basili and R. Perricone. Software Errors and Complexity: An Empirical Study. *Communications of the ACM*, 27(1):42–52, January 1984.
4. T. Budd. Mutation analysis: Ideas, examples, problems and prospects. In *Computer Program Testing*, pages 129–148, Amsterdan: North Holland, July 1981.
5. M. Chaim. Poke-tool — Uma Ferramenta para Suporte ao Teste Estrutural de Programas Baseado em Análise de Fluxo de Dados. Master's thesis, DCA/FEE/UNICAMP, Campinas, SP – Brazil, 1991. In Portuguese.
6. R. DeMillo, R. Lipton, and F. Sayward. Hints on Test Data Selection: Help for the Practicing Programmer. *Computer*, 11(4), April 1978.
7. R. A. DeMillo and A. P. Mathur. On the Use of Software Artifacts to Evaluate the Effectiveness of Mutation Analysis for Detecting Errors in Production Software. In *Thirteenth Minnowbrook Workshop on Software Engineering*, July 1990.
8. G. Denaro and M. Pezzè. An Empirical Evaluation of Fault-Proneness Models. In *ICSE2002 – International Conference on Software Engineering*, pages 241–251, Orlando, FL, May 2002.
9. N. E. Fenton and M. Neil. A Critique of Software Defect Prediction Models. *IEEE Transactions on Software Engineering*, 25(5):675–689, September/October 1999.
10. A. Fitzsimmons and T. Love. A Review and Evaluation of Software Science. *ACM Computing Surveys*, 10(1):3–18, March 1978.
11. J. R. Gaffney. Estimating the Number of Faults in Code. *IEEE Transactions on Software Engineering*, 10(4), 1984.
12. M. H. Halstead. *Elements of Software Science*. Elsevier, North-Holland, 1975.
13. A. Koenig. *C Traps and Pitfalls*. Addison-Wesley, 1988.
14. M. Lipow. Number of Faults per Line of Code. *IEEE Transactions on Software Engineering*, 10(4), 1982.
15. T. J. McCabe. A Complexity Measure. *IEEE Transactions on Software Engineering*, 2(4):308–320, December 1976.
16. K. H. Moeller and D. Paulish. An Empirical Investigation of Software Fault Distribution. In *First Int'l Software Metrics Symposium*, pages 82–90. IEEE CS Press, 1993.
17. S. Rapps and E. Weyuker. Data flow analysis techniques for test data selection. In *ICSE1982 – International Conference on Software Engineering*, pages 272–278, Tokio - Japan, September 1982.
18. S. Rapps and E. Weyuker. Selecting software test data using data flow information. *IEEE Transactions on Software Engineering*, SE-11(4), April 1985.
19. E. Wong, J. R. Horgan, S. London, and A. Mathur. Effect of Test Set Minimization on Fault Detection Effectiveness. *Software-Practice and Experience*, 28(4):347–369, April 1998.
20. E. Wong, J. R. Horgan, S. London, and A. P. Mathur. Effect of Test Set Minimization on the Fault Detection Effectiveness of the All-Uses Criterion. Technical Report tr152p, Purdue University, 1994.