Exploring defect correlations in a major Fortran numerical library

Tim Hopkins Les Hatton*

February 21, 2008

Abstract

We analyse a mature numerical Fortran software library comparing its defect history against a wide variety of well-known static measurements searching for significant correlation. We found that most of these static measurements individually correlate moderately well with a high degree of significance with the number of defects in a particular component but principle component analysis reveals that defect prediction is influenced by most to some modest extent but no particular factor dominated. The analysis revealed a number of interesting properties however including the existence of negative correlations which may suggest that programmers take more care in certain scenarios and less care in others.

We go on to show that when static measurements are averaged across components, strong evidence of a logarithmic relationship with defect emerges.

Keywords: Defects, Numerical software, software metrics

1 Overview

The ability to predict future program failures from static properties of programs (i.e. properties which can be directly measured from the source code) has long been a goal of software engineering researchers. Efforts based on

*Computing Laboratory, University of Kent T.R.Hopkins@kent.ac.uk and CISM, Kingston University, L.Hatton@kingston.ac.uk

predicting future failures based on dependence on error-prone language features (for example mechanisms which lead to the loss of significant bits) have generally proven fruitful, [14]. Unfortunately, efforts attempting to find satisfactory correlation of program failure with structural properties such as the number of decisions, are undermined by the very disparate nature of software with many programming languages in use based on numerous different paradigms. The resulting representation differences make failure modelling erratic to say the least, [3]. Some recent efforts have produced significant results for structural metrics in OO systems, [17], but all such efforts rely on the availability of high quality failure records over a period of time coupled with access to source code and excellent version control. Such opportunities do not arise very often and here we are able to analyse one such dataset acquired over a number of years and present the results using standard statistical tests of significance as a contribution to the empirical base of this subject. The large number of components measured give much confidence in the results.

In the interests of both pedagogy and repeatable science, the sanitised raw data are freely available for download and analysis [1] in the form of a zipped Excel spreadsheet.

1.1 The analysis of a numerical software library

The NAG (Numerical Algorithms Group) Library[5] is a very widely-used set of scientific procedures. Over the last thirty years, they have been continually enhanced to keep pace with research in numerical analysis and have also spread from the original implementation language of Fortran 66 and 77 into other languages such as C, Ada and Fortran 95. Here we analyse the Fortran 77 library over a number of releases which provides an excellent opportunity to study defect growth for several reasons:-

- The package has a complete and carefully maintained defect history which was embedded in program headers and for which perl scripts to mine the header defect information could easily be designed.
- The package is large, (266,123 executable lines of code (XLOC) as analysed in 3659 subroutine/functions),
- As is often the case with software experiments, no usage or coverage data was available but the data shown here covers a period of more

[1] http://www.leshatton.org/Defect_Correlations_05-02-2008.html

than 20 years, a relatively long maturity time and the defect density is likely to be more asymptotically representative.

- The package is of good quality for its generation (1978-) and difficult application area with an asymptotic defect density of 4.9/KXLOC (thousand executable lines of code).

1.2 Extraction of static measurements

The complete source code of the library was made available to us and we designed and implemented parsing tools for the full Fortran 77 language. This turned out to be necessary in order to be able to extract all the desired static code measurements. The parsing engine was designed using handcrafted lexical and syntactical analysers to cater for:-

- The generally non-significant behaviour of the space character in Fortran 77, (the first lexical step is to discard all spaces outside strings or in the first 5 character positions of a Fortran line).

- The arbitrary nature of the look-ahead in Fortran necessary to resolve grammatical structures such as the I/O implied DO loop.

The code measurements, often known as metrics, were chosen on the basis of their common occurrence in the literature or anecdotally over the years. As a result, 15 properties were extracted for the 3659 components. The five letter codes after each item header will be used as abbreviations for the corresponding parameter throughout the rest of the paper. We will continue to refer to these as parameters rather than metrics in order to preserve conventional mathematical definitions of a metric.

1. *Knot count: STKNT.* A knot is a crossing of control flow as illustrated for example in [15]. Knots only occur in languages which have explicit non-structural jump constructs such as the eponymous goto statement in its various forms. The goto statement is a necessary but not a sufficient condition for a knot as it can be used to simulate nested (knot-free) structures as well as non-nested structures. The existence of knots is often referred to informally as 'spaghetti' code.

2. *Cyclomatic complexity: STCYC.* A graph theoretic measurement which is essentially a count of the number of decisions as first introduced by [11].

3. *Extended cyclomatic complexity: STMCC.* An extension to the cyclomatic complexity based on complex predicates introduced by [12]. A complex predicate contains either logical disjunctive ('or') or conjunctive ('and') phrases or both.

4. *Maximum level of nesting of IF statements: STMIF.* This is included for anecdotal reasons. It is thought to be associated with testing difficulties.

5. *Number of declared objects actually used: STVAR.* This is included for anecdotal reasons.

6. *Number of subroutines in a file: STSUB.* This is included for anecdotal reasons, however, it should be noted that in Fortran 77, unlike C, the file has no special linguistic meaning.

7. *Number of executable lines of code: STXLN.* Executable lines of code counts the number of lines which generate executable code when compiled. Many defect models have been built using executable lines of code as an independent variable. Note that Fortran continuation lines were not counted.

8. *Number of backward jumps: STBAK.* This is included for anecdotal reasons. It is thought to interfere with readability.

9. *Number of dangling elseifs, i.e. an if .. else if with no else clause: STELF.* This is included for anecdotal reasons and is believed to indicate the presence of incomplete logical thought.

10. *Number of gotos: STGTO.* There has been so much discussion of this over the years since the initial comments of [2], that we felt we could not leave it out. In addition, Fortran 77 has a rich set of goto forms including the arithmetic IF and the absence of any form of WHILE construct means that goto statements in various forms are used unusually frequently.

11. *Number of undeclared variables: STUNV.* There has again been largely anecdotal attribution that this indicates some level of sloppiness and may therefore be related to defect.

12. *Length of shortest, and longest jump via a goto: STLJM, STHJM.* The rationale behind this is, once again, anecdotal.

13. *Logarithm of the path count: STLPT.* The path count is the number of ways through a particular program assuming that all paths are equally likely, [6]. The rationale behind it is that it is more sensitive to decision complexity than the cyclomatic number, (for example it can distinguish between a sequential series of if statements and a single switch statement containing the same number of clauses, which have the same cyclomatic complexity). This is similar to the NPATH metric of Nejmeh, [13].

14. *Total number of operator tokens: STOPT.* The rationale behind this is, once again, anecdotal. More details can be found in [15].

2 Statistical analysis

We chose to approach this in an exploratory way initially by using principle component analysis on normalised parameter data to characterise the relationship between the above 15 parameters and the general shape of the data cloud to see if there were obviously dominant parameters. Normalisation corrects for the very different scales observed for the parameters, for example, executable lines of code might run up to 2000 or so whilst the maximum depth of IF nesting might only be 6.

We followed this up by using linear regression analysis again on normalised data, of defect occurrence with respect to selected parameter pairs leading to a full multiple linear regression analysis including only those parameters which are statistically significant in the regression, [16].

Finally, to overcome any noise induced by different levels of usage, a non-linear analysis was performed on averaged data.

2.1 Principle Component Analysis

Of the initial 15 parameters, 9 including somewhat surprisingly the number of executable lines, were rejected early on as having a p-value greater than 0.05, a standard criterion for rejecting parameters in such analysis.

Rejection of executable lines of code Since many previous experiments try to model the number of defects as a function of the number of executable lines of code, we felt it worthwhile exploring this in a little more detail. The most likely reason for rejection of this seemed to be that there was

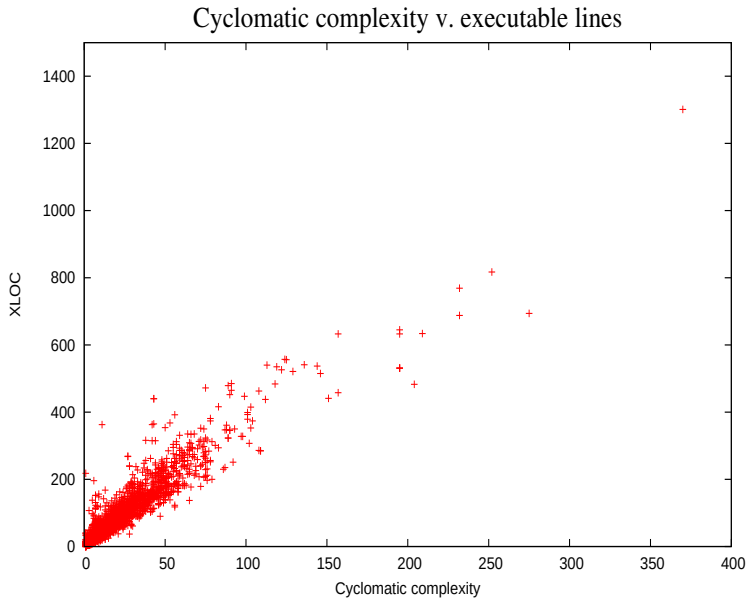

Figure 1: A scatter diagram between the number of executable lines and the cyclomatic complexity. A high correlation is clearly visible.

already a parameter included in the regression model with which it was highly correlated. A short analysis revealed that cyclomatic complexity is very highly correlated with executable lines of code as shown in Figure 1 yielding a regression equation of

$$XLOC = 10.0 + 3.64v(G) \tag{1}$$

where v(G) is the cyclomatic complexity with associated p-values for both the constant and v(G) around 0.0. in essence, this equation states that it is extremely likely that there will be a decision about every 3-4 executable lines in a typical program in this library, a perhaps not unsurprising observation. This has been noted before by [18] but on a much smaller dataset of only 26 routines.

The principle component analysis on the 6 remaining significant parameters revealed the eigenvalues shown in Figure 2. In essence this means that re-aligning the axes using linear combinations of these parameters is able to account for most of the shape of the observed data cloud and the biggest eigenvalue corresponds to that linear combination which is in the direction of the largest variability. As is often the case in principle component analysis, one eigenvalue tends to dominate and the component corresponding to

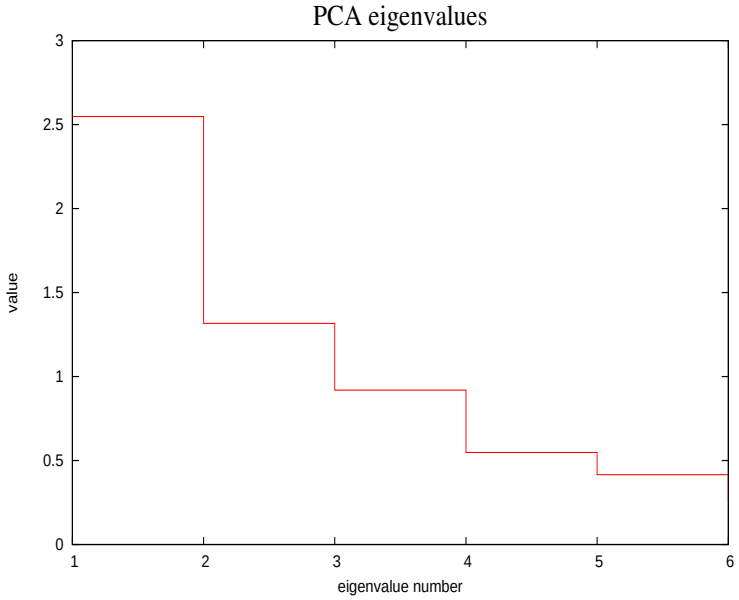

Figure 2: The eigenvalues resulting from principle component analysis

this highest eigenvalue is shown in Figure 3. This reveals a confusing picture however, with no single candidate from the original parameters emerging as being in any way dominant in defining the principle direction of the data cloud.

2.2 Pairwise linear correlations with defects

The confused nature of the relationship between significant parameters and the direction of the data cloud can be seen for example in Figures 5 and 6 which show scatter plots of *defects v. number of goto statements* and *defects v. the logarithm of the path count*. In each case, correlations are relatively weak although significant as indicated by the following statistic, [16]

- Test of Hypothesis $\rho = 0$, where ρ is the theoretical population correlation coefficient. This uses the fact that:-

$$t = \frac{r(N-2)^{0.5}}{(1-r^2)^{0.5}} \tag{2}$$

  is distributed according to Student's t-distribution with (N-2) degrees of freedom, where r is the sample product-moment correlation coefficient

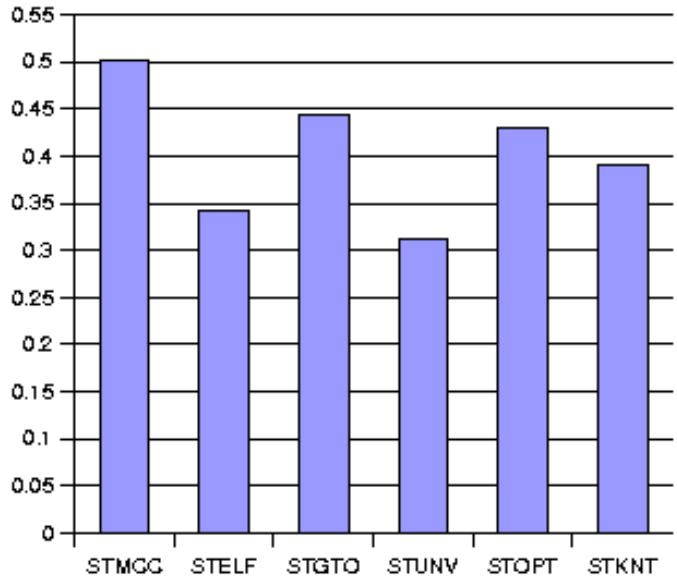

Figure 3: The individual parameter contributions to the principle component

$$r = \frac{\sum_{i=1}^{N} x_i y_i}{((\sum_{i=1}^{M} x_i^2)(\sum_{i=1}^{N} y_i^2))^{0.5}} \tag{3}$$

where $x_i = X_i - \overline{X}$ and $y_i = Y_i - \overline{Y}$ in standard notation.

For 1% significance, the t-value must be at least 2.33. All of the parameters tried give values of t in excess and sometimes greatly in excess of 10.

2.3 Pairwise linear anti-correlations with defects

Of potentially more interest, some of the parameters turned out to be negatively correlated with defect. Although the correlation is not strong, it is also statistically significant and these seem rather harder to explain. For example, figure 7 shows a scatter plot of defects v. maximum depth of nested IF statements. Here there is a modest but clear negative correlation which again turns out to be significant at the 1% level. One possible explanation for this is that nested IF statements are inherently more complex to deal with logically and that programmers automatically take more care in such situations. However another possibility is that covering deeply nested statements may be more difficult implying that defects in deeply nested code are

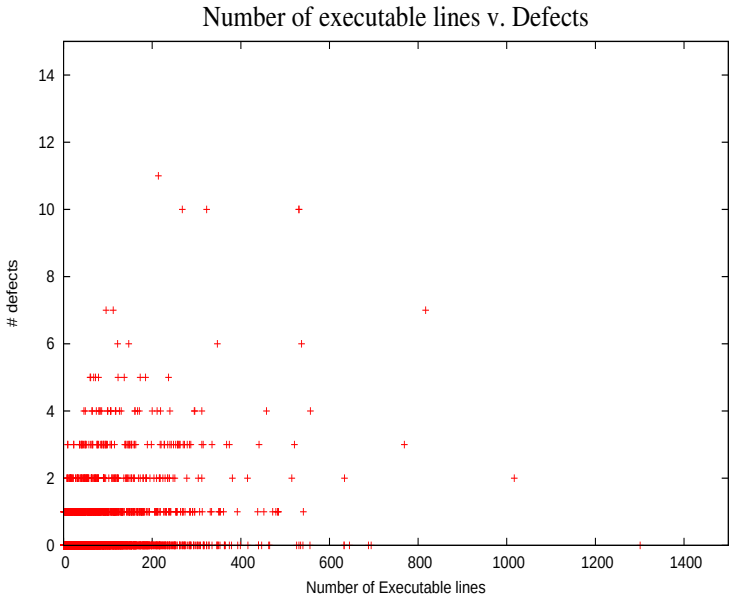

Figure 4: A scatter diagram between the number of defects and the number of executable lines.

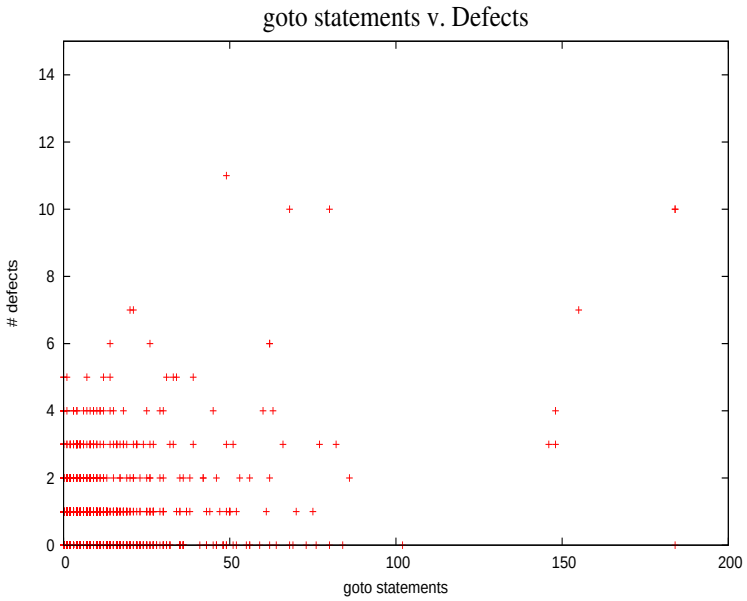

Figure 5: A scatter diagram between the number of defects and the number of goto statements.

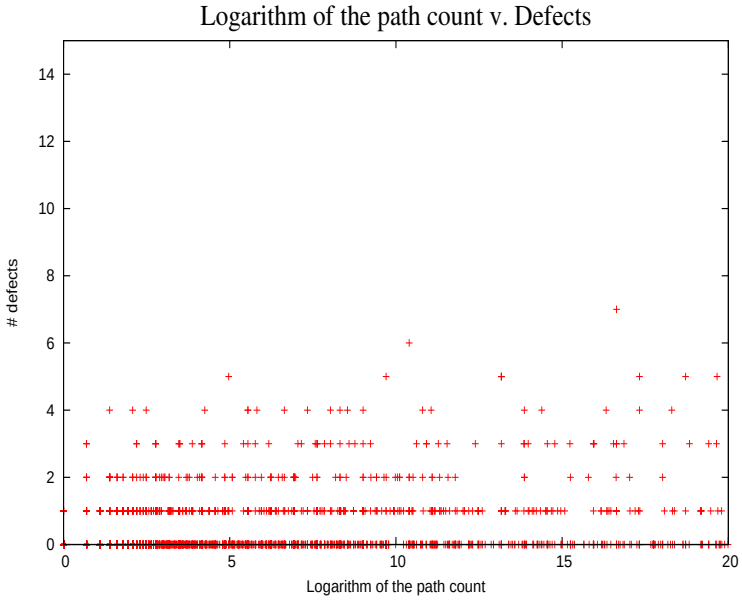

Figure 6: A scatter diagram between the number of defects and the logarithm of the path count.

harder to illuminate.

There is also statistically significant but again moderate anti-correlation between the number of dangling *else if* statements and the number of defects at the 5% level. This is in clear opposition to rules such as those proposed in influential programming standards such as the automotive industry's MISRA C 2004 standard, (rule 14.10, [10], which bans dangling *else if* statements on the grounds that they lead to an increased risk of defect, although it does not justify this). It is impossible from the data to cast more light on this case unfortunately.

2.4 Multiple linear regression analysis

For completeness, we built a multiple linear regression model for the number of defects as a function of the 6 components whose contributions were highly significant in that they all had p-values considerably less than 0.05. the resulting model is

$$\begin{aligned} Defects = {} & 0.837 STMCC \\ & -1.29 STELF + 5.03 STGTO \\ & +0.555 STUNV + 0.445 STOPT + 2.263 STKNT \end{aligned} \tag{4}$$

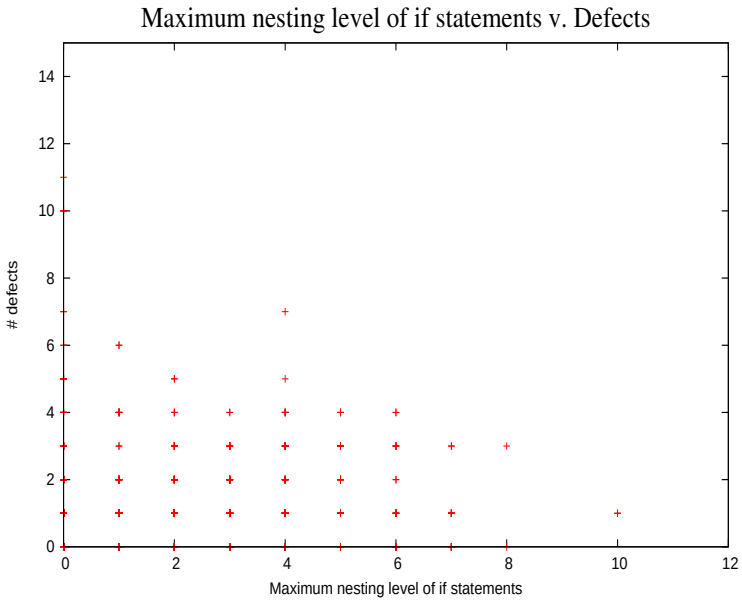

Figure 7: A scatter diagram between the number of defects and the maximum level of nesting of IF statements.

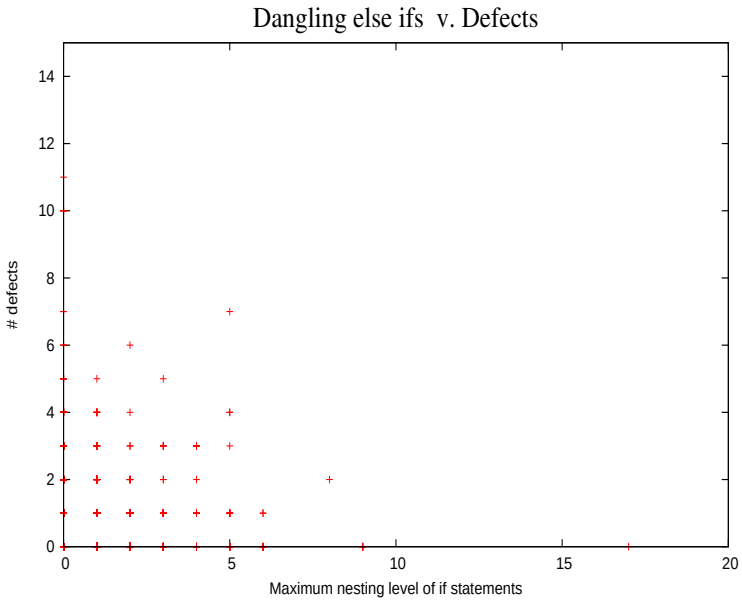

Figure 8: A scatter diagram between the number of defects and the number of dangling else if statements.

This model is significantly different from models which attempt to describe defects as some functional dependence on executable lines of code alone, for example, [9], [4], [1] or [7] and is based purely on statistically significant multiple linear regression of the raw data.

To close this section, we will assess the consistency of the current dataset against the logarithmic growth of defect described by [9], [7] and [8] amongst others.

2.5 Non-linear modelling

As was mentioned earlier, one of the problems with software defect data analysis is there is generally no idea of how much a particular component has been used. Even for mature systems such as this one, it is perfectly possible for a component to have very little usage and therefore simply not have enough time to present a representative defect profile. This can be circumvented to a certain extent by computing the average size of component associated with each number of defects. When this is done on the current dataset, strong evidence of logarithmic behaviour is present not just relative to the executable lines of code but also relative to some of the other static measurements also as can readily be seen in Figure 9. Note that there were insufficient components with more than 7 defects to calculate a statistically reliable mean so the data is truncated at 7 defects. The maximum number of defects in any component was 11.

Finally some authors, [9], have modelled defects in terms of STXLN log(STXLN). Intriguingly, there is evidence of zones of linearity in this dataset also when this behaviour is used as can be seen in Figure 10.

3 Discussion and Conclusions

A large and widely used mature scientific subroutine library has been analysed in a variety of ways to identify possible predictive relationships between static properties of the code (i.e. measurable at compile time), and defects which were subsequently found in use.

Through linear analysis, we conclude

- Principle Component Analysis and Multiple Linear Regression reveal a confused picture. Although most of the parameters were weakly but

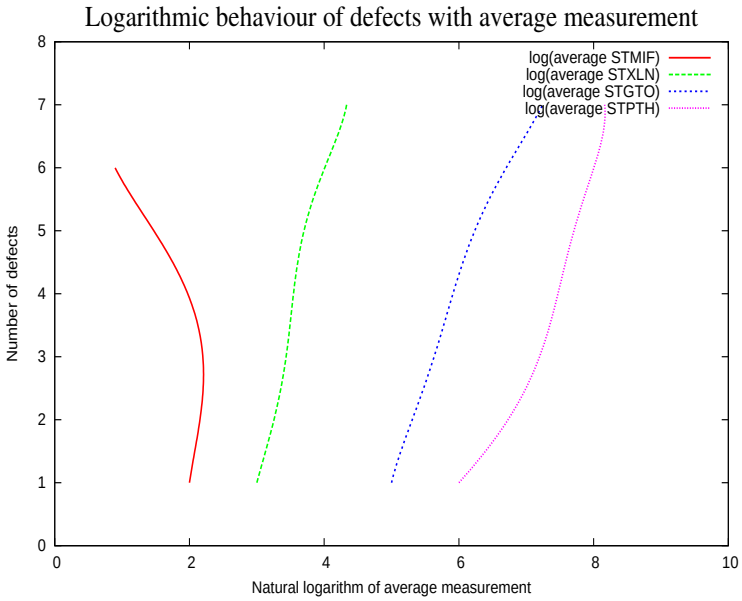

Figure 9: Illustrating the generally linear behaviour of defects with the logarithm of the average of various measurements. The negative linear trend with increasing defect for the logarithm of the average of the maximum level of IF nesting (STMIF) can be clearly seen in comparison with the positive linear trend exhibited by each of the others displayed. This was also noted in the linear analysis.

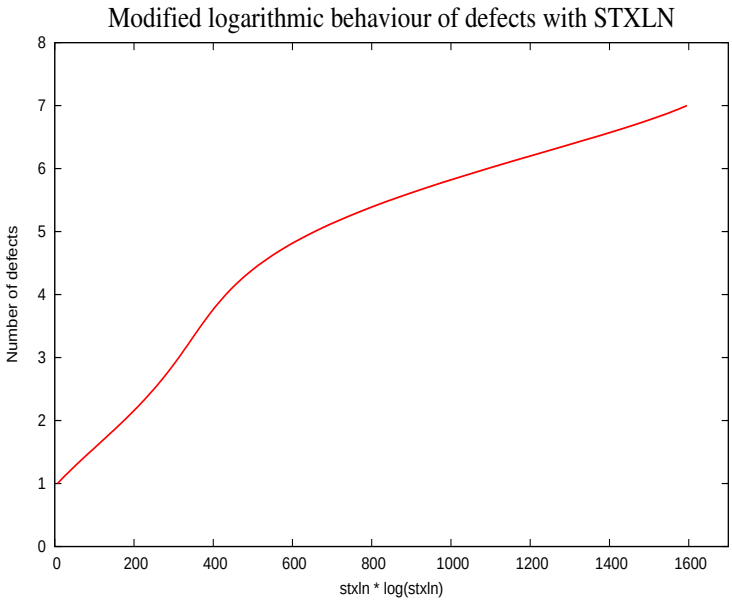

Figure 10: The number of defects plotted against STXLN log(STXLN). Zones of linearity are clearly visible.

statistically significantly correlated with the number of defects, none is particularly dominant in determining the defect behaviour. Divining reliable predictive relationships between the expected number of defects and structural parameters such as those presented here does not look particularly promising based on this dataset.

- Two interesting and statistically significant anti-correlations were observed. First, the number of defects appeared to be anti-correlated to the maximum depth of nesting, possibly implying that programmers take more trouble when implementing such code, although this would require further experimental data to clarify. Second, the number of defects was anti-correlated with the number of dangling else if statements in clear contradiction to rules proposed in influential standards such as [10] which prohibit the dangling else if because of its supposed propensity for inducing defect. It seems clear that further defect investigations on major, mature packages such as that studied here might have much to say on the relevance of a number of common practices and beliefs.

Finally, non-linear analysis demonstrated that when values of particular measurements are averaged across all components to reduce the effect of

potential usage differences, strong evidence of logarithmic behaviour with defect, (positive and in some cases negative), emerged. In particular, the logarithmic relationship between defects and executable lines of code noted by authors such as [9], [7] and [8], amongst others, is well supported.

References

[1] B.T. Compton and C. Withrow. Prediction and control of ada software defects. *Journal of Systems and Software*, 12:199–207, 1990.

[2] E.W. Dijkstra. Go to statement considered harmful. *Comm. ACM*, 11(3):147–148, 1968.

[3] N.E. Fenton and M. Neil. A critique of software defect prediction models. *IEEE Transactions on Software Engineering*, 25(5):675–689, 1999.

[4] J.R. Gaffney. Estimating the number of faults in code. *IEEE Transactions on Software Engineering*, 10(4), 1984.

[5] Numerical Algorithms Group. NAG Fortran library, 1978–1999. http://www.nag.com/.

[6] L. Hatton. *Safer C: Developing software in high-integrity and safety-critical systems*. McGraw-Hill, 1995. ISBN 0-07-707640-0.

[7] L. Hatton. Re-examining the fault density v. component size connection. *IEEE Software*, 14(2):89–98, 1997.

[8] A. Gunes Koru, Dongsong Zhang, and Hongfang Liu. Modeling the effect of size on defect proneness for open-source software. In *PROMISE '07: Proceedings of the Third International Workshop on Predictor Models in Software Engineering*, page 10, Washington, DC, USA, 2007. IEEE Computer Society.

[9] M. Lipow. Number of faults per line of code. *IEEE Transactions on Software Engineering*, 8(4):437–439, 1982.

[10] MIRA Ltd. Guidelines for the use of the programming language C in critical systems, 2004. http://www.misra.org.uk/.

[11] T. McCabe. A software complexity measure. *IEEE Transactions on Software Engineering*, 2(4):308–320, 1976.

[12] G.J. Myers. An extension to cyclomatic measure of program complexity. *SIGPLAN Notices*, 12(10):61–64, 1977.

[13] B.A. Nejmeh. Npath: A measure of execution path complexity and its applications. *Comm. ACM*, 31(2):188–200, 1988.

[14] S.L. Pfleeger and L. Hatton. Do formal methods really work ? *IEEE Computer*, 30(2):p.33–43, 1997.

[15] M.L. Shooman. *Software Engineering*. McGraw-Hill, 2nd edition, 1985.

[16] M.R. Spiegel and L.J. Stephens. *Statistics*. Schaum. McGraw-Hill, 3rd edition, 1999.

[17] R. Subramanyam and M.S. Krishnan. Empirical analysis of CK metrics for object-oriented design complexity: Implications for software defects. *IEEE Transactions on Software Engineering*, 29(4):297–310, April 2003.

[18] M.R. Woodward, M.A. Hennell, and D. Hedley. A measure of control flow complexity in program text. *IEEE Transactions in Software Engineering*, 5(1):45–50, 1979.