Computer Science at Kent

Defect Patterns and Structural Properties in a Mature Well-Specified Software System

Tim Hopkins

Les Hatton, Kingston University.

Technical Report No. 5-08
Date

Published by the Computing Laboratory,
University of Kent, Canterbury, Kent CT2 7NF, UK

Defect patterns and structural properties in a mature well-specified software system

TIM HOPKINS
University of Kent, UK
and
LES HATTON
Kingston University, UK

Software engineering is not an empirically based discipline. As a result, many of its practices are based on little more than a generally agreed feeling that something may be true. Part of the problem is that it is both relatively young and unusually rich in new and often competing methodologies. As a result, there is little time to infer important empirical patterns of behaviour before the technology moves on. Very occasionally an opportunity arises to study the defect growth and patterns in a well-specified software system which is also well-documented and heavily-used over a long period.

Here we analyse the defect growth and structural patterns in just such a system, a numerical library written in Fortran evolving over a period of 30 years. This is important to the wider community for two reasons. First, the results cast significant doubt on widely-held long standing beliefs and second, some of these beliefs are perpetuated in more modern technologies. Since we obviously generalise from older languages to new, it makes good sense to use empirical long-term data when it becomes available to re-calibrate those generalisations. At the same time, the results contain intriguing glimpses into defect behaviour which may transcend whatever technology is in use.

Categories and Subject Descriptors: D.2.8 [**Software Engineering**]: Metrics—*Complexity Metrics*; D.2.2 [**Software Engineering**]: Design Tools and Techniques—*Software libraries*

General Terms: Measurement
Additional Key Words and Phrases: Correlations, defects, numerical software, principal component analysis, 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 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, see

Authors' addresses: Tim Hopkins, Computer Science Department, University of Kent, Canterbury, Kent CT2 7NF, UK; email: T.R.Hopkins@kent.ac.uk; Les Hatton, CISM, Kingston University, Penrhyn Road, Kingston-upon-Thames, Surrey KT1 2EE, UK; email: L.Hatton@kingston.ac.uk

Pfleeger and Hatton [1997]. Unfortunately, efforts attempting to find satisfactory correlation of program failure with structural properties such as the number and type of decisions, are undermined by the very disparate nature of software with many programming languages in use based on numerous different paradigms.

Despite this, some beliefs have become surprisingly well-entrenched across numerous languages in a wide range of application areas. For example, in spite of significant evidence to the contrary (see [Fenton and Neil 1999]) cyclomatic complexity [McCabe 1976], a graph theoretic measure essentially related to the number of decisions in computer programs, has been widely used in many disparate developments as a predictor of components which will be unreliable in some sense. The danger of its continuing use is that it might become a design criterion for limiting components to a maximum cyclomatic number as is the case in the recent Joint Strike Fighter C++ standard, a safety-critical environment, ([Joint Strike Fighter 2005, AV Rule 3]).

Even longer standing is the debate over the *goto* statement. The result of this debate initiated by Dijkstra [1968], which was unconstrained by any relevant measurement at the time, was that the *goto* statement is to this day believed to be strongly correlated with program failure in whatever context it arises, including its implicit forms such as the *continue* and *break* statement which appear in numerous modern languages. This appears in such influential standards as both editions of MISRA C ([MIRA Ltd. 1998, Rule 56], [MIRA Ltd. 2004, Rule 14.4]), the Joint Strike Fighter C++ standards, [Joint Strike Fighter 2005, AV Rule 189], and the European Space Agency Ada standard [European Space Agency 1998]. Other language independent code fragments which have invited opprobrium include the so-called dangling *else if*, being an *if .. else if ..* clause with no *else* statement ([MIRA Ltd. 2004] and [Joint Strike Fighter 2005]), and also restrictions on the maximum depth of nesting of control structures, [European Space Agency 1998].

It is clear then that these beliefs are well-entrenched in modern development. However, as we show here, none of these appear to have any statistically significant basis in fact when studied over a long period in a well-documented and well-specified system *in which such effects should have appeared if they had any substance.* We do not seek to rehabilitate the *goto* statement or indeed any of the programming fragments which have attracted negative comment over the years. However, we do strongly emphasise the importance of empirical evidence in supporting claims if we are to understand the essence of software engineering in order to improve its rather erratic history of systems failure. That such beliefs still appear in a modern context and are accepted without empirical challenge is sufficient motive for the present work. The nature of design and component interactions does indeed evolve with time as programming languages implement more or less of such paradigms as OO, but all programs must make decisions and such decisions are often tainted with beliefs formulated in the distant past without much intervening influence from measurement.

Not only are some practices condemned without measurement but others are similarly supported without measurement. Although some recent efforts [Subramanyam and Krishnan 2003] have produced interesting results for structural metrics in OO systems, other empirical studies [Hatton 1998] suggest that some of the

suggested benefits of such systems may be rather more illusory than was hoped. However, 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 an unusually long period 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 while the relevance of these results to the wider community is unquestionable for the reasons outlined above.

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 [NAG 1999] 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 three decades, 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 onwards) and covers a difficult application area with an asymptotic defect density of 4.9/KXLOC (thousand executable lines of code).

—The package is unusually well-specified for a software system as it implements procedures defined in mathematical notation. It is therefore effectively free of the problems which occur in many systems through imprecisely defined requirements.

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 hand-crafted lexical and syntactical analysers to cater for

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

—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, by Shooman [1985]. 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 McCabe [1976].

(3) *Extended cyclomatic complexity: STMCC.* An extension to the cyclomatic complexity based on complex predicates introduced by Myers [1977]. 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 Dijkstra [1968], 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 [Hatton 1995]. 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 put forward by Nejmeh [1988].

(14) *Total number of operator tokens: STOPT.* The rationale behind this is, once again, anecdotal. More details can be found in Shooman [1985].

2. STATISTICAL ANALYSIS

Here we study the correlation between and amongst defects and parameters associated with some of the more commonly occurring beliefs using regression analysis.

2.1 Cyclomatic complexity versus executable lines of code

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 zero. 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 also been noted very recently and emphatically across different languages by van der Meulen [2008] on a very much larger sample. Here we simply confirm the observation that the cyclomatic complexity appears to add no additional significant information to that already contained in the count of executable lines and they are effectively interchangeable.

2.2 Pairwise linear correlations with defects

As we have shown above, a number of existing beliefs which have found their way into modern programming standards for safety-critical systems relate to assertions about a positive correlation between defect and the appearance of a particular programming construct such as the eponymous *goto* statement, or the presence of *if .. else if* statements without an *else* clause, (the so-called dangling else construct), or the cyclomatic number. In this study however, the confused nature of the relationship between nearly *any* of the significant parameters measured here and the

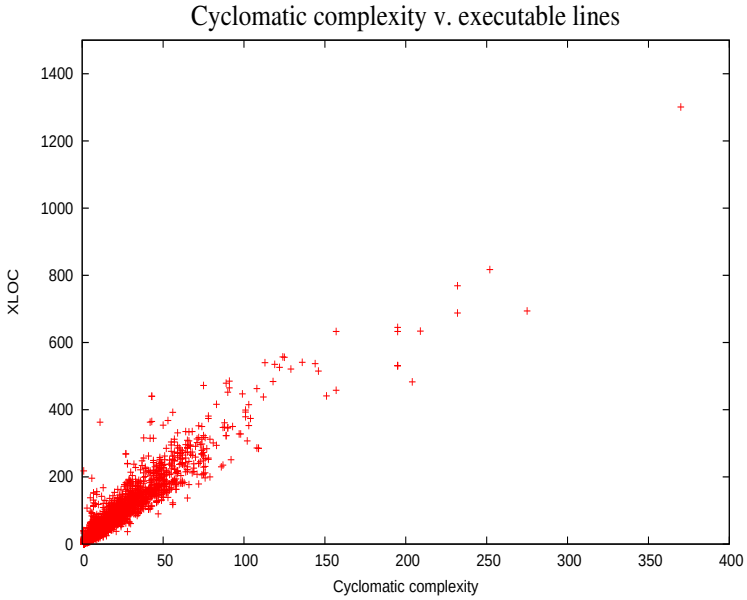

Fig. 1. A scatter diagram between the number of executable lines and the cyclomatic complexity. A high correlation is clearly visible.

growth of defect data is exemplified by Figure 2 which shows a scatter plot of *defects against number of goto statements*. This graph is typical of the unsmoothed relationship between defects and most parameters. In each case, using the t-test on the product-moment correlation coefficient confirms that, although significant at the 1% level, the correlations are weak at best [Spiegel and Stephens 1999].

2.3 Pairwise linear anti-correlations with defects

Other long-standing beliefs of positive correlation turned out to be negatively correlated in the current study.

For example, it was mentioned above that defects are believed to be associated with the appearance of the dangling *else if* construct described above and various of the quoted standards continue to mitigate against their use. This rule also occurs in many of the standards described in Hatton [2004]. In fact, the opposite appears to be true in the current study where there is modest *anti*-correlation between the number of dangling *else if* statements and the number of defects which is statistically significant at the 5% level.

Another example is shown in Figure 3 which shows a scatter plot of defects against maximum depth of nested *if* statements. Again, in a number of the standards described in European Space Agency [1998] and Hatton [2004], advice is given to avoid deeply nested decision statements. Here, however, there is a modest but clear anti-correlation which turns out to be significant at the 1% level. It is unclear what the mechanism for this is, but one possible explanation 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

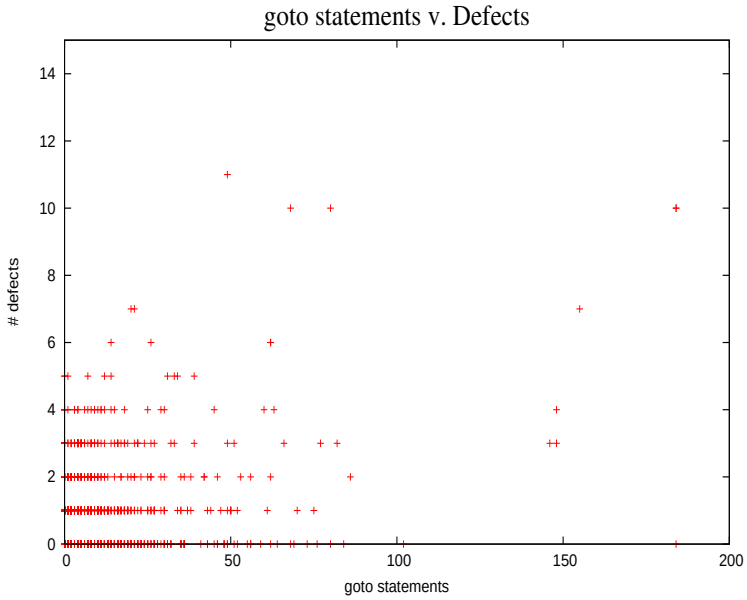

Fig. 2. A scatter diagram between the number of defects and the number of goto statements.

covering deeply nested statements may be more difficult implying that defects in deeply nested code are harder to illuminate. Either way, the data is not compatible with the advice often provided.

Given the confused nature of these simple relationships, we then chose to expand this approach using Principle Component Analysis (PCA) on normalised parameter data to see if we could identify any more emphatic relationship between the 15 parameters measured and the general shape of the defect data cloud. In particular, we were interested to see if there were any obviously dominant parameters.

Essentially, PCA constructs a data cloud using these 15 independent variables. Such data clouds are rarely isotropic implying that some parameters are more important than others in fitting the observed defect growth. PCA excludes parameters with no effect automatically and rotates the data cloud amongst the remaining parameters looking for principle directions or *components* which match the observed shape of the defect growth well. The data were normalised to correct 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.

2.4 Principle Component Analysis

Of the initial 15 parameters, 9 were rejected early on as having a p-value greater than 0.05, a standard criterion for rejecting parameters in such analysis. We have already explained that cyclomatic complexity is so highly correlated with executable lines of code that they are effectively indistinguishable so of the two, cyclomatic complexity was temporarily retained here.

PCA on the 6 remaining significant parameters revealed the eigenvalues shown

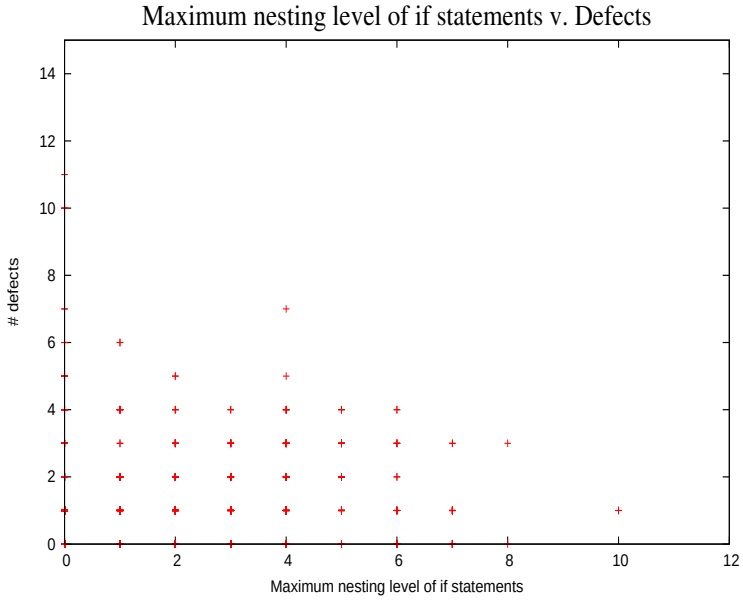

Fig. 3. A scatter diagram between the number of defects and the maximum level of nesting of *if* statements.

Eigenvalue number	Value
1	2.55
2	1.32
3	0.92
4	0.55
5	0.42
6	0.25

Table I. The dominant eigenvalues of the Principle Components.

in Table I.

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. It is often the case in principle component analysis that one eigenvalue tends to dominate as we have here. However the corresponding eigenvector contains relatively equal contributions from each parameter as can be seen in Table II. In other words, we have a confusing picture with no single candidate from the original parameters emerging as being in any way dominant in defining the principle direction of the data cloud.

We conclude that *in the NAG dataset, there is no simple linear relationship in the raw data between single parameters and defect growth which could justify the nature of the rules given in the coding standards described earlier even though the dataset is of sufficient maturity that by now, any such phenomenon, if present, would be expected to have manifested itself significantly.* Instead, the predominant

Parameter	Normalised contribution
STCYC	0.50
STELF	0.34
STGTO	0.44
STUNV	0.31
STOPT	0.43
STKNT	0.39

Table II. The contributions to the principle eigenvector from each parameter.

variability in the data cloud is aligned along a direction which is defined by a complex combination of several parameters which would appear to defy any attempt to render into a simple rule.

Given this degree of noise, an obvious avenue to explore is whether smoothing reveals any underlying subtle pattern.

2.5 Smoothed data

Over the years, there has been much interest shown in relating the number of defects to the number of lines of code used (see, for example, Lipow [1982], Hatton [1997] and Koru et al. [2007]). As was mentioned earlier, one of the problems with software defect data analysis is that 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 had 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 in executable lines of code associated with each number of defects. Such data is inevitably very noisy but the dataset under study here is so large that when this is done, strong evidence of logarithmic behaviour emerges. The logarithmic behaviour appears to take the form of $d \sim x \log x$ where d is the number of defects recorded in a component and x is the number of executable lines as can be seen in Figure 4, which shows significant zones of linearity. This functional behaviour has previously been recorded by Lipow [1982]. The departure at lower values of $x \log x$ is interesting and will be commented on shortly. 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, although the maximum number of defects in any component was 11.

3. POWER-LAW BEHAVIOUR IN COMPONENT SIZES

There is current interest in the observed distribution of component sizes in systems, (see, for example, [Potanin et al. 2005] and [Hatton 2008]), so evidence for this behaviour was also sought. A power-law distribution has the general form

$$p(s) = \frac{k}{s^{\alpha}} \tag{2}$$

where $p(s)$ is the probability that a certain size s will appear. On a $\log p - \log s$ scale, this is a straight line with negative slope. However, as noted by Potanin et al. [2005], there are some drawbacks in using this with discrete systems, as rare events

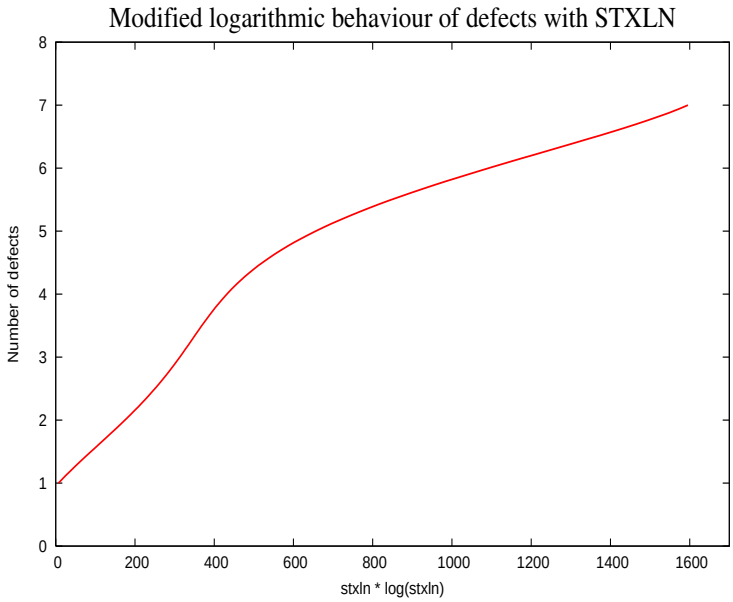

Fig. 4. The number of defects plotted against STXLN log(STXLN). Zones of linearity are clearly visible.

are quite noisy in the following sense. There might, for example, be component sizes of 1000 lines and 1500 lines in a particular system, with no component sizes in between. To circumvent this, it is common practice to look for power-law behaviour in *rank* ordering rather than size and this is what we did here.

The relevance of this is (as Hatton [2008] demonstrates using an argument based on statistical mechanics) that an *a priori* power-law distribution of component sizes (i.e., where components are rank ordered as frequency against size, the distribution obeys an inverse power law) will automatically lead to an $x \log x$ distribution of defects in a software system as it approaches maturity, i.e., enters a quasi-equilibrated state, although the approximations used in the mathematical development tend to degrade for small x. The question arises as to whether such power-law behaviour is present from the beginning, i.e., a property of the design stage, or whether it emerges as software systems tend to an equilibrium state. Although our data here does not stretch back to the very first release of this library, there appears to be very little difference in the component size distribution with different releases over many years even though the library grew by a factor of two in this period as Figure 5 illustrates. In other words such power-law behaviour appears to be present from a very early stage in the evolution of the library. The work of Hatton [2008] then implies that the $x \log x$ behaviour demonstrated for larger x in Figure 4 is inevitable and independent of the implementation details.

4. DISCUSSION AND CONCLUSIONS

A large and widely used mature scientific subroutine library has been analysed to test a number of widely-held beliefs about the relationship between defects and

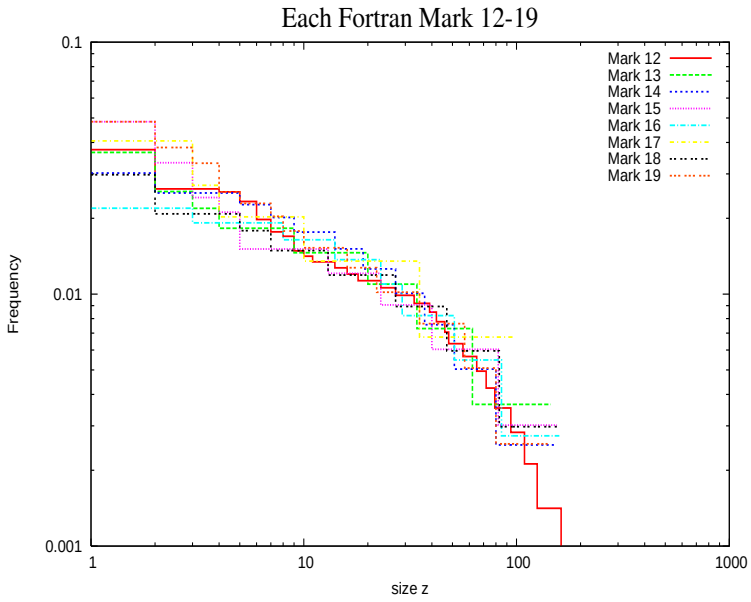

Fig. 5. The distribution of component sizes displayed as the probability of a certain component size appearing as a function of its rank-ordering plotted as a log − − log display as is normal for power-law behaviour.

either the structural properties of the code or its language features. The relevance of this to the wider community is that such beliefs have been held for many years and are widely applied in modern programming standards even in critical systems, and may therefore influence the failure behaviour of such systems. Such beliefs include the relationship of defects with the *goto* statement, dangling *else* constructs, maximum levels of decision nesting, component size and a number of other code features which occur frequently in programming standards for all languages.

We conclude the following

—Principle Component Analysis shows that no single measurement parameter of the 15 given here is dominant in determining the defect behaviour. In particular, the belief that the *goto* statement is strongly related to defects is not upheld by this study and the cyclomatic complexity number appears to have little if any utility as a predictor of defect over and above that contained in a count of the executable lines of code.

—Some patterns which are commonly believed to be positively correlated with defect, such as the presence of the dangling *else* statement, turn out to be anti-correlated as does the maximum level of decision nesting.

—When the data is averaged over component size, defects appear to be broadly proportional to $x \log x$ where x is the number of executable lines of code although the data is still noisy.

—Power-law behaviour of component size appears across all versions of the library considered here, spanning more than 30 years. The unchanging nature of this

whilst the library continues to grow strongly suggests that this behaviour appears naturally at the design stage. In other words, it appears to have been present from the very beginning.

We stated, at the beginning of this paper, that a number of long-held beliefs have found their way into modern standards for development across multiple languages, even for safety-critical systems. Such beliefs inevitably influence the way that developers produce systems so it is of some considerable importance to underpin them with empirical support wherever possible. In the present study on a mature system over many years, a number of those beliefs if supported should have left identifiable traces at some level of significance. They have not.

We conclude therefore that the beliefs are probably erroneous. This is just one study but it is very large, covers a long period and was carried out in an application area which was unusually well-specified implying that the resulting defect data is of higher precision than in less well-specified areas. Perhaps the biggest single lesson therefore is that beliefs unsupported by any empirical evidence are fundamentally unreliable and if we are to make progress in avoiding failures with modern technologies, only those methodologies soundly based on empiricism are likely to be of any lasting help.

REFERENCES

Dijkstra, E. 1968. Go to statement considered harmful. *Comm. ACM 11,* 3, 147–148.

European Space Agency. 1998. Ada coding standard. ftp://ftp.estec.es.nl/pub/wm/wme/bssc/bssc983.pdf.

Fenton, N. and Neil, M. 1999. A critique of software defect prediction models. *IEEE Transactions on Software Engineering 25,* 5, 675–689.

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

Hatton, L. 1997. Re-examining the fault density v. component size connection. *IEEE Software 14,* 2, 89–98.

Hatton, L. 1998. Does OO sync with the way we think ? *IEEE Software 15,* 3, 46–54.

Hatton, L. 2004. Safer language subsets: an overview and a case history, MISRA C. *Information and Software Technology 46*, 465–472.

Hatton, L. 2008. Power-law distributions of component sizes in general software systems. *IEEE Transactions on Software Engineering*. Submitted for publication - second review.

Joint Strike Fighter. 2005. Air vehicle C++ coding standards. http://www.jsf.mil/downloads/documents/.

Koru, A. G., Zhang, D., and Liu, H. 2007. 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*. IEEE Computer Society, Washington, DC, USA, 10.

Lipow, M. 1982. Number of faults per line of code. *IEEE Transactions on Software Engineering 8,* 4, 437–439.

McCabe, T. 1976. A software complexity measure. *IEEE Transactions on Software Engineering 2,* 4, 308–320.

MIRA Ltd. 1998. Guidelines for the use of the programming language C in vehicle based systems. http://www.misra.org.uk/.

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

Myers, G. 1977. An extension to cyclomatic measure of program complexity. *SIGPLAN Notices 12,* 10, 61–64.

NAG. 1978–1999. NAG Fortran library. http://www.nag.com/.

Nejmeh, B. 1988. NPATH: A measure of execution path complexity and its applications. *Comm. ACM 31,* 2, 188–200.

Pfleeger, S. and Hatton, L. 1997. Do formal methods really work ? *IEEE Computer 30,* 2, p.33–43.

Potanin, A., Noble, J., Frean, M., and Biddle, R. 2005. Scale-free geometry in OO programs. *Comm. ACM. 48,* 5 (May), 99–103.

Shooman, M. 1985. *Software Engineering*, 2nd ed. McGraw-Hill, London.

Spiegel, M. and Stephens, L. 1999. *Statistics*, 3rd ed. Schaum. McGraw-Hill, London.

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

van der Meulen, M. 2008. The effectiveness of software diversity. Ph.D. Thesis, City University, London.