Unifying power-law behaviour, functionality and defect distribution in general software systems

Les Hatton
CISM, University of Kingston*

November 28, 2008

Abstract

In a previous paper, the close link between power-law distribution of component sizes and defect growth in maturing software systems, independently of their representation language, was revealed by the use of statistical mechanical arguments. It was further noted that power-law distributions appear to be present in software systems from the beginning implying that xlogx behaviour of defect growth where x is the number of executable lines of code in a component, is by far the most likely outcome. Strong evidence of this behaviour was presented.

Building on the above work, this paper presents a theoretical model which is able to predict the observed *a priori* appearance of power-law behaviour in component sizes in developing systems using another variational principle linked with a modification of the Hartley/Shannon information content, therefore unifying the observed phenomena under twin variational principles, one appropriate to the development phase and one to the release phase.

Keywords: Defects, Macroscopic system behaviour, Component size distribution, Power-law

*L.Hatton@kingston.ac.uk, lesh@oakcomp.co.uk

1 Background

In [3], a model based on statistical mechanics was presented which showed that in a software system subject only to the twin constraints of total size and number of defects present *a priori*, that a component size distribution exhibiting power-law behaviour was inextricably linked with a growth in defects within a component of xlogx where x is the number of executable lines of code.

For reference, power-law behaviour can be represented by the probability p(x) of a certain size x appearing being given by a relationship like:-

$$p(x) = \frac{k}{x^{\alpha}} \tag{1}$$

which on a log p - log x scale is a straight line with negative slope. The size x will be measured here in executable lines of code. For software systems, such distributions are usually plotted in rank order as explained by [7]. This tends to reduce the noise and is exact provided size is a power of rank.

Power-law behaviour was demonstrably present in two large scientific subroutine libraries written in C and Fortran of considerable longevity with excellent audit trails of development as reported by [3] along with numerous other systems in several languages and various sizes in the range 2,000 - 250,000 lines of code. In addition, the presence of asymptotic xlogx behaviour of defect, where x is the number of executable lines of code, was demonstrated in the same libraries as exemplified by Figure 1, extracted from the C library data.

In that paper, although one implied the other, the question arose as to which if any is the driver. Does power-law behaviour force xlogx defect evolution or the other way round ? The paper went on to present emphatic evidence that power-law behaviour in component-size appears to have been present from the early days following the release of a large software system as is exemplified by Figure 2. Although Figure 2 does not show the first third approximately of the released life-cycle because the data was not available, the data is neverthless compelling. Even more emphatic evidence for *a priori* power-law behaviour is presented from a very different system written in a very different language (Tcl-Tk) as represented in Figure 3. This diagram shows the development of a graphical user interface of approximately 40,000 lines of code across all 44 revisions from its first appearance in 2003 to the

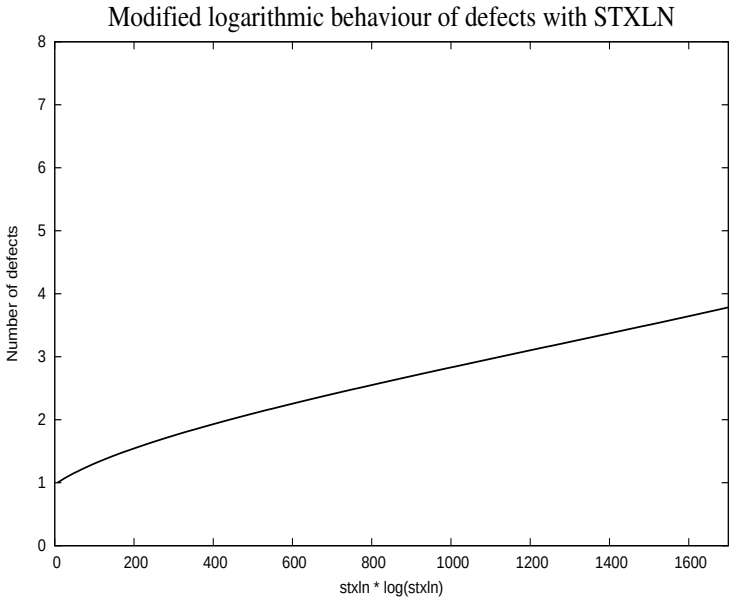

Figure 1: The number of defects in components of a mature and widely-used numerical C library as a function of xlogx where x (shown as the code STXLN in the diagram) is the number of executable lines of code as reported by Hatton and Hopkins, [4]. These data cover a period of 15 years and therefore present a good example of a quasi-equilibriated system in which the linearity is striking.

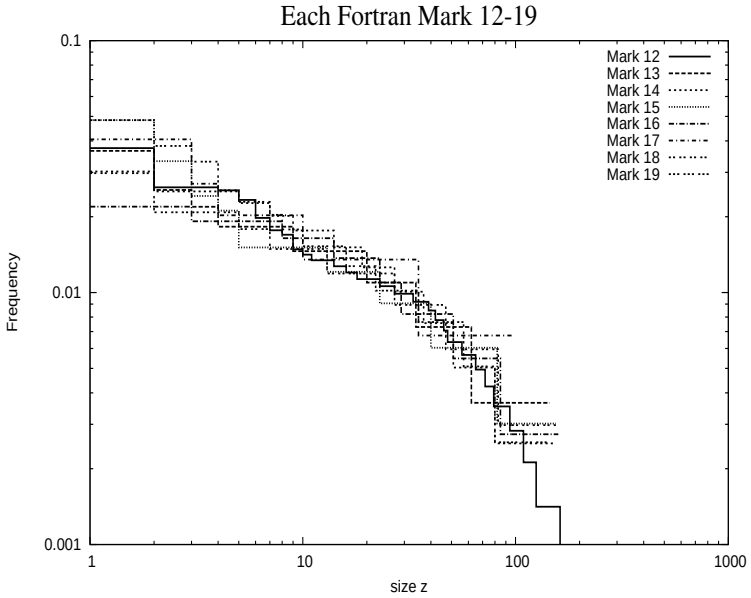

Figure 2: The characteristic power-law behaviour in the distribution of component sizes in various versions of a major Fortran numerical library as reported by [4]. This data covers the development from about a third of the way into the life-cycle up to the present day.

present day, (only every 4th version is shown). Note that Figures 2, 3 and 4 are show rank order although the same behaviour is present in the size order as described in [3].

In addition, there is very strong evidence for the appearance of power-law behaviour generally in many systems independently of their representation language as exemplified by Figure 4 and also in the work of [7] who demonstrated the same phenomenon in OO systems. [3] then shows that this leads to this result:-

> *As a released software system of a fixed size approaches a quasi-equilibrium state in the sense that defects present from the beginning are simply reported as they appear, an* ***a priori*** *power-law distribution of component size will inevitably imply that the most likely defect distribution in a component will asymptotically evolve as xlogx, where x is the number of executable lines of code. This behaviour is independent of implementation details.*

The paper then concluded by leaving the reason why power-law behaviour in component sizes should be present from the earliest days of a system for

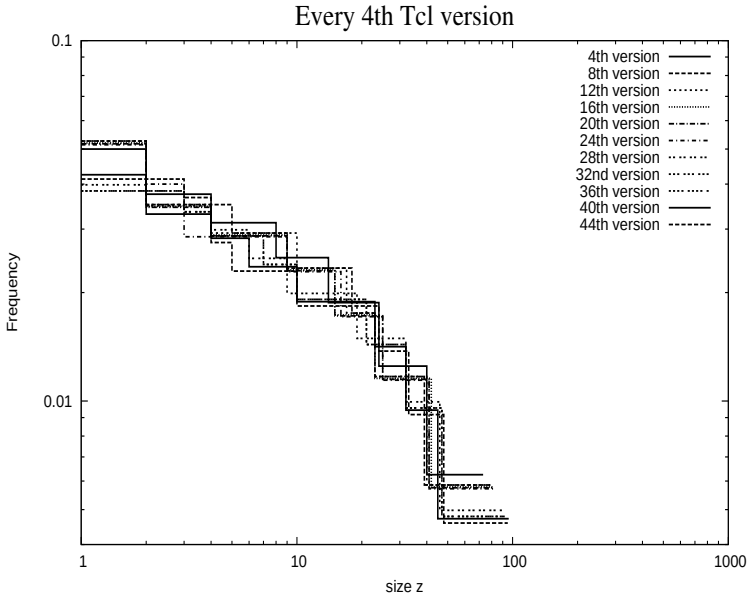

Figure 3: The characteristic power-law behaviour in the distribution of component sizes in the first 44 versions of a GUI development written in Tcl-Tk and covering some 4 years of development.

further study.

This paper builds on [3] by developing a physically reasonable model to show why power-law behaviour in component size should be present in software systems from the earliest days of a released software system, thus linking the two pieces of work by twin applications of the same variational principle in different contexts: one based on information theory in the development phase and one based on the growth of defect in the released phase. Finally, it shows that the two are unified by a simple relationship between functionality and defect.

2 Functionality

Following the same approach as in [3], suppose that a system is made up of M components each of size n_i executable lines of code such that the total size is given by

$$N = \sum_{i=1}^{M} n_i \tag{2}$$

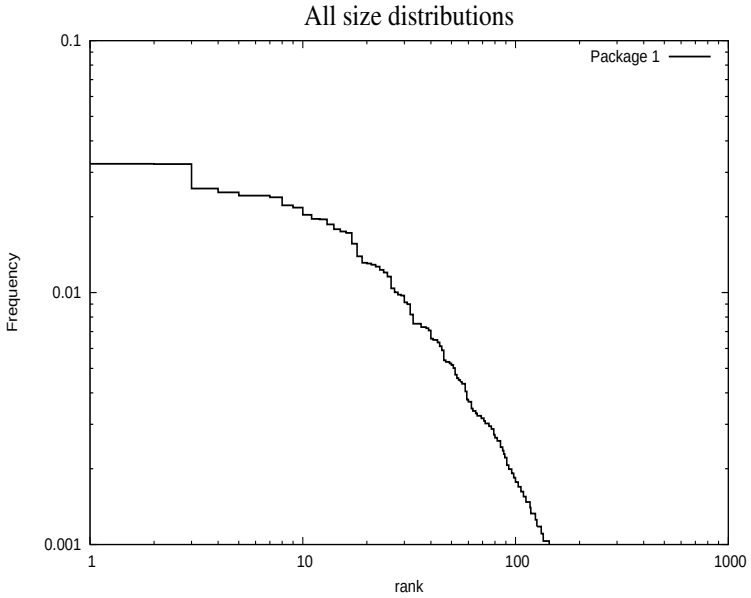

Figure 4: Power-law behaviour in the distribution of component sizes across many systems as reported by [3].

Suppose there is also some externally imposed entity ε_i associated with each line of component i whose total amount is therefore given by

$$U = \sum_{i=1}^{M} n_i \varepsilon_i \tag{3}$$

Using the method of Lagrangian multipliers, the following variational will be maximised

$$logW = NlogN - \sum_{i=1}^{M} n_i log(n_i) + \gamma\{N - \sum_{i=1}^{M} n_i\} + \beta\{U - \sum_{i=1}^{M} n_i \varepsilon_i\} \tag{4}$$

where γ and β are the multipliers. Setting $\delta(logW) = 0$ leads to

$$0 = -\sum_{i=1}^{M} \delta n_i \{log(n_i) + \alpha + \beta \varepsilon_i\} \tag{5}$$

where $\alpha = 1 + \gamma$. This must be true for all variations δn_i and so

$$log(n_i) = -\alpha - \beta \varepsilon_i \tag{6}$$

Using equation (2) to replace α, this can be manipulated into the most likely, i.e. the equilibrium distribution

$$n_i = \frac{Ne^{-\beta\varepsilon_i}}{\sum_{i=1}^{M} e^{-\beta\varepsilon_i}} \tag{7}$$

Following [8] and defining $p_i = \frac{n_i}{N}$ and referring to equation (3), p_i can be interpreted as the probability that a component gets a share of U equal to ε_i. Manipulating equation (7) then yields

$$p_i = \frac{e^{-\beta\varepsilon_i}}{\sum_{i=1}^{M} e^{-\beta\varepsilon_i}} \tag{8}$$

In other words, the probability of finding a component with a large amount of ε_i is correspondingly small. Given the externally imposed nature of ε_i, p_i can be taken to be the probability that a component of n_i lines actually occurs.

So far this is a completely standard development as followed in [8] and [3] for example.

Now the concept of *functionality* will be introduced. At this stage, it will be construed as having an obvious meaning related to the functional requirements of the system in the following sense. Suppose that the i^{th} functional component has associated with it a functionality f_i. The total functionality F is therefore given by

$$F = \sum_{i=1}^{M} f_i \tag{9}$$

The same computational device used in [3] will now be applied again. Noting that equation 9 can be written as

$$F = \sum_{i=1}^{M} n_i(\frac{f_i}{n_i}) \tag{10}$$

leads directly to the identification of ε_i with $(\frac{f_i}{n_i})$ in equation (7). In other words, each line of component i has a functionality density associated with it given by $(\frac{f_i}{n_i})$. Note that introducing this functional dependence of ϵ_i on n_i does not disrupt the development which led to equation (5) as the introduced term is $O(\frac{1}{n_i^2})$.

Equation (8) can then be written as

$$p_i = \frac{e^{-\beta \frac{f_i}{n_i}}}{Q(\beta)} \tag{11}$$

where

$$Q(\beta) = \sum_{i=1}^{M} e^{-\beta \frac{f_i}{n_i}} \tag{12}$$

3 Functionality and information

Historically, functionality has proven to be an elusive goal to quantify for computer scientists as evidenced by the fact that lines of code still appears to be the dominant measure of *size*, even though a *line of code* is itself a somewhat arbitrary measure, strongly related to the implementation language and also the developer's personal taste. In addition, different alternatives present themselves, for example, in C and C++, the following are all used

- SLOC, (source lines of code). Simply a measure of the count of lines as seen by a text editor.
- PPLOC, (pre-processed lines of code). SLOC contain comments and some argue that they should not be counted. In C and C++, the pre-processor removes comment in a predictable way but also expands header files leading to a definition of PPLOC as a count of the number of pre-processed non-blank lines of code.
- XLOC, (executable lines of code). A count of those lines of source code which cause the compiler to generate executable code. This is the preferred measure here as it is rather less dependent on the nature of the programming language than either of the first two.

Although these are all different measures, they are usually highly correlated as exemplified by Figure 5, so knowledge of any one can be used to predict the others with good accuracy.

The relationship of any line of code measure with functionality is much harder to understand however.

Because of these difficulties, other alternatives have been proposed to capture the amount of functionality in a system such as the *function point*. However, these also present difficulties as shown by [5] and [6].

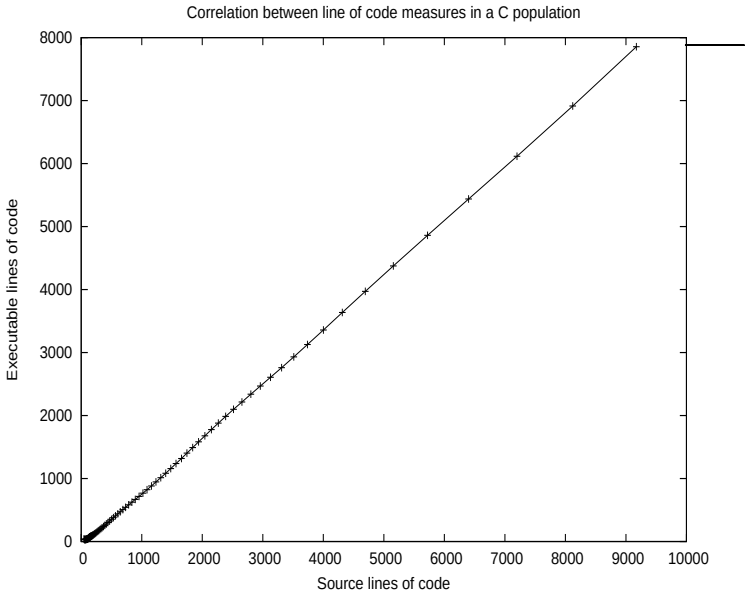

Figure 5: The correlation between SLOC and XLOC in a population of around 300,000 SLOC of C programs.

Since there appears to be no obvious measure and since also this is a statistical approach, lines of code will be used as a generic measure of size and the relationship between functionality and *information content* will be explored as described by [2], developed by [9] and contextually related in the seminal book of [1].

Hartley in [2] showed that a message of N signs chosen from an *alphabet* or code book of S signs has S^N possibilities and that the *quantity of information* is most reasonably defined as the logarithm of the number of possibilities. The obvious analogy for a component of size n_i lines each containing on average k symbols of a programming language of total alphabet S(n_i) is to define the quantity of information as

$$I(n_i) = logS^{kn_i}(n_i) = kn_i logS(n_i) \tag{13}$$

Here, S is allowed to be a function of n_i as it is reasonable to expect that the alphabet of available signs may depend on the size of the component being built. However, it should be noted as discussed by [1], p. 50, that the concept of information here as extended by Shannon and Wiener amongst others, only relates to the symbols themselves and not their *meaning*. This is not satisfactory for the current model as it does not distinguish between

pounding aimlessly on a keyboard and the careful construction of a computer program to specific intent.

The definition which will be used therefore follows from the extraordinary, and to be brutally frank, serendipitous observation that the number of non-redundant user-defined variable names in a program appears *on average* to be linearly related to the size in executable lines of code as will be shown now.

The raw data is somewhat unprepossessing as can be seen by inspecting Figure 6 which shows the relationship between the number of non-redundant user-defined variable names and number of executable lines of code in each component of the NAG Fortran scientific subroutine library as described in [4]. (The largest 15% have been excluded as outliers to be consistent with Figure 7 as the bins defined there are not so well populated for large numbers of user-defined variable names.) A non-redundant variable name is one which is actually used in the program. However the consistent theme between this work and [3] has been a statistical mechanical approach, i.e. that macroscopic principles might well emerge from the massive microscopic complexity of a software system in the same way that thermodynamical relations emerge from statistical mechanics for complex physical systems.

Motivated by the form of equation (11), if the data of Figure 6 is averaged such that the number of non-redundant user-defined variables is gathered into bins in multiples of 5 and then plotted against the average count of executable lines of code for that bin, then Figure 7 emerges. This extraordinary relation shows that in this mature and very large library, the number of non-redundant user-defined variable names in a component is *on average* linearly proportional to the count of executable lines of code.

A definition of functionality This leads to the following definition of functionality, (for reasons which will become obvious shortly).

> *The functionality of a component is defined to be the Hartley/Shannon information content of the alphabet S, of non-redundant user-defined variable names used to construct that component.*

Using equation (13), this can be written as

$$f_i = logS^{kn_i}(n_i) = kn_i logS(n_i) = kn_i log\eta n_i \tag{14}$$

where η is some constant, exploiting the linearity of the relationship shown in Figure 7.

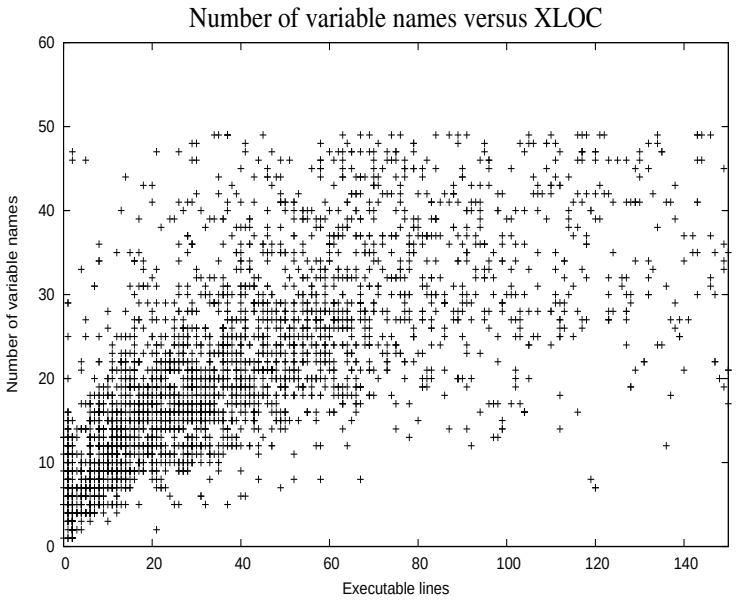

Figure 6: The raw data for the NAG Scientific Subroutine Library. Here the non-redundant variable name count for each component is plotted against the number of executable lines of code in that component.

This nicely circumvents the criticism that the standard Hartley/Shannon quantity of information in a message constructed from an alphabet takes no account of the meaning of the symbols in that alphabet. Here, the very fact that the developer has used them associates meaning with them in a fundamental way. The fact that the built-in symbols of a programming language are also excluded, for example keywords such as *if*, *while*, also reduces the dependence on stylistic attributes of a language, (there are many equivalent ways of constructing the same *if* statement even in the same programming language as will be demonstrated shortly). Equation (11) can then be written

$$p_i = \frac{e^{-\mu log S(n_i)}}{Q(\beta)} \tag{15}$$

Combining equations (15) and (14) then gives

$$p_i = \frac{e^{-\mu log \eta n_i}}{Q(\beta)} \tag{16}$$

Equation (16) then leads directly to a prediction that component sizes will be distributed as a power-law

$$p_i = \propto n_i^{-\delta} \tag{17}$$

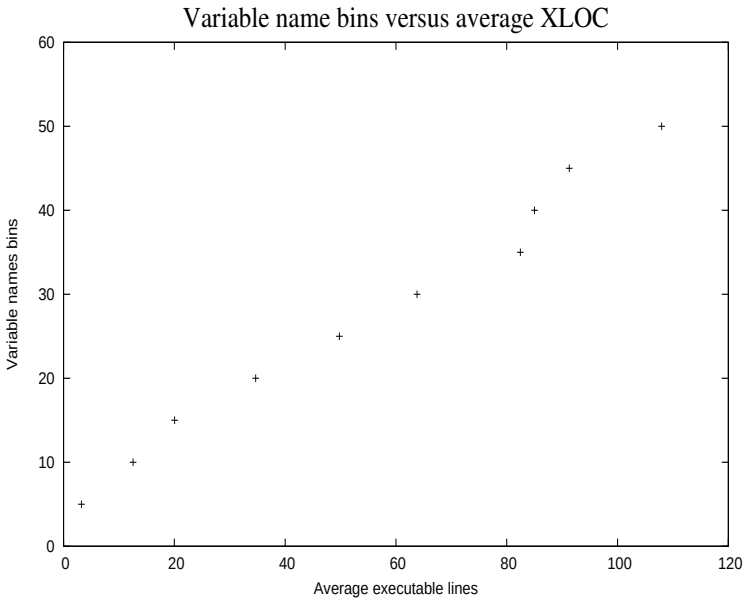

Figure 7: A markedly linear relationship emerges in the NAG Fortran Scientific Subroutine Library when the number of non-redundant user-defined variable names is binned in multiples of 5 with each bin plotted against the average executable line of code count for that bin.

where δ is another constant.

These steps are worth summarising.

> *In the development of a software system of fixed size and fixed functionality where functionality is identified directly with the Hartley/Shannon information content of the non-redundant user-defined variable names of a component, coupled with the observation that those variable names grow linearly with size, the most likely distribution of component sizes in the developed system will be as a power-law.*

4 Functionality and robustness

Before bringing things together, it is appropriate briefly to revisit functionality and the relationship with the measure of size used here which is executable lines of code, along with the notion of symbols from an alphabet in Hartley/Shannon information. One of the main drawbacks of using lines of code as a size measure can be illustrated as follows. Consider the following program fragments of identical functionality written in the language C:-

1. ```
   max_of_xy = (y>x) ? y : x;
   ```

2. ```
   max_of_xy = x;
   if (y>x) max_of_xy = y;
   ```

3. ```
   max_of_xy = x;
   if (y>x)
      max_of_xy = y;
   ```

4. ```
   if (y>x)
      max_of_xy = y;
   else
      max_of_xy = x;
   ```

5. ```
   if (y>x) {
      max_of_xy = y;
   } else {
      max_of_xy = x;
   }
   ```
   ```

Here, precisely the same functionality is displayed in 1,2,3,4 and 5 line fragments illustrating not only the natural variability of the developer but also the redundancy present in typical programming languages. However, the number of occurrences of user-defined variable names is much less variable in these fragments (5 in the first and 6 in each of the others), supporting the use of an alphabet based on user-defined names as being a more robust definition of symbols in real programs. Given that on average, executable lines of code appear to be linearly related to the number of non-redundant user-defined variable names, it is worthwhile asking how robust the development described in this paper is to scale changes in the executable line of code count for whatever reason.

Consider a new measure of line of code to be given by

$$n_i = \lambda n'_i \tag{18}$$

Substituting equation (18) in equation (17) reveals immediately that the predicted scale-law behaviour is insensitive to linear changes of scale in counting lines of code.

Given that other code measures such as a count of non-redundant user-defined variable names as described above or the cyclomatic complexity as shown in [4] have an essentially linear relationship with executable lines of code when averaged, this implies that power-law behaviour should also emerge if one of these is used as measure of size. However, it also illustrates that even with natural scale changes due to variability of implementation, the number of executable lines of code *on average* is a perfectly reasonable measure of code size and probably as good as anything else.

5 Conclusions

It is appropriate to unite the conclusions from [3] and the current work in the following way even though chronologically they occurred in the opposite order.

First of all, [3] was motivated by considering the role of defect first as it appears in an already developed system. This included examples of power-law behaviour being present from the earliest days in released systems of different sizes and implementation languages. In view of its pivotal role in [3], a model was sought and found in this paper to offer an explanation

as to why such power-law behaviour should be present from the beginning. Combining the results of both papers, the following model for the whole life-cycle therefore emerges:

Development phase The current paper demonstrates the following:-

> *In the development of a software system of fixed size and fixed functionality where functionality is identified directly with the Hartley/Shannon information content of the non-redundant user-defined variable names of a component, coupled with the empirical observation that those user-defined variable names grow linearly with size, the most likely distribution of component sizes in the developed system will be as a power-law.*

Further evidence was presented for the presence of power-law behaviour throughout the life-cycle of typical systems and also for the linear relationship between the number of non-redundant user-defined variable names and the number of executable lines of code.

Release phase When such a system is placed in the users' hands, it will contain a fixed (and generally unknown) number of defects. As shown by [3], with a matching variational principle,

> *After release, as a software system matures and gradually exhibits the defects inadvertently built in during its development, an underlying power-law distribution of component sizes will lead in a system of fixed size to a distribution in defect in each component of xlogx where x is the number of executable lines in that component.*

Significant evidence in favour of the appearance of both power-law behaviour in component size distribution in numerous systems in different languages and also the xlogx asymptotic behaviour of defect as systems age was presented accompanied this conclusion.

To summarise, [3] and the current paper therefore present a consistent statistical mechanical model of the complete life-cycle of a software system independently of its implementation in which macroscopic properties such as a power-law distribution of component sizes and the xlogx component defect behaviour as components age, naturally emerge from simple assumptions about the constraints under which such systems are built.

A further contribution in the present paper is made in finding a consistent definition of functionality of a component as the Hartley/Shannon information content of the alphabet used to construct programs consisting solely of the non-redundant user-defined variable names allowing unification of the two variational principles: one in the development phase for functionality and one in the release phase for defect. This does not complete the story on information content in computer programs by any means as there may be other empirical measures of functionality which lead to the same power-law behaviour of component size, but this will be postponed for the moment to further study of these fascinating relationships.

Finally, it can be noted that the instincts of the long-suffering end user seem always to have been correct in associating bugs with features. The variational principles used in [3] and the current paper are entirely consistent if defects are linearly proportional to functionality for any definition of functionality which has a linear relationship with the number of executable lines of code. This can easily be seen by comparing the association of functionality density here, and the defect density in [3], with the entity ε_i in the unifying variational principle represented by equation (8).

With this view of quantity of information in a software system therefore, defects *are* functionality.

6 Acknowledgements

The author would like to acknowledge useful discussions on power-law behaviour in general physical systems with his colleague Dr. Miro Novak and also the unknown reviewers of the first paper, who helped clarify the arguments in significant ways.

References

[1] Colin Cherry. *On Human Communication.* John Wiley Science Editions, 1963. Library of Congress 56-9820.

[2] R.V.L. Hartley. Transmission of information. *Bell System Tech. Journal*, 7:535, 1928.

[3] L. Hatton. Power-law distributions of component sizes in general software systems. *IEEE Transactions on Software Engineering*, 2008. Accepted for publication.

[4] T.R. Hopkins and L. Hatton. Defect correlations in a major numerical library. *Submitted for publication*, 2008. Preprint available at http://www.leshatton.org/NAG01_01-08.html.

[5] Barbara Kitchenham. Counterpoint: The problem with function points. *IEEE Software*, 14(2):29,31, 1997.

[6] Barbara Kitchenham, Shari Lawrence Pfleeger, Beth McColl, and Suzanne Eagan. An empirical study of maintenance and development estimation accuracy. *J. Syst. Softw.*, 64(1):57–77, 2002.

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

[8] P.K. Rawlings, D. Reguera, and H. Reiss. Entropic basis of the pareto law. *Physica A*, 343:643–652, July 2004.

[9] C.E. Shannon. A mathematical theory of communication. *Bell system technical journal*, 27:379,423, July 1948.