Power-laws, persistence and the distribution of information in software systems

Les Hatton
CISM, University of Kingston*

January 12, 2010

Abstract

In a previous paper, an intimate link between power-law distribution in the tail of component sizes and defect appearance in maturing software systems, independently of their representation language, was revealed by the use of a variational method built on statistical mechanical arguments.

Building on the above work, this paper first of all demonstrates experimentally that power-law behaviour in the tail of component sizes appears to be a persistent property in that it is present from the earliest release of software systems. It then develops a model for this behaviour by combining statistical mechanics with a modification of Hartley/Shannon information content to predict power-law behaviour in the alphabet of tokens used in programming languages; the approximate linearity of user-defined variable names with component size, and finally, that defects will also obey a power-law, leading directly to a prediction of defect clustering. Each of these is supported by good experimental evidence across numerous systems. The model therefore unifies the distribution of information content through the development process with the appearance of power-law behaviour in alphabet and consequently, component size as well as the distribution of defects in the release phase.

Keywords: Information content, defect clustering, Component size distribution, Power-law

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

1 Background

In [10], 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*, a component size distribution exhibiting power-law behaviour in medium to large components (i.e. greater than about 20 executable lines of code (XLOC)) was inextricably linked with a growth in defects within a component of xlogx where x is the number of XLOC. Numerous systems in several different languages were also analysed to show that such power-law behaviour in component size was indeed present. The paper concluded by saying that it was not however clear which if either of these two phenomena was the driver.

The current paper builds on this work and makes a number of further contributions.

- First, it demonstrates using further experimental evidence that power-law behaviour in the tail of component sizes in disparate software systems is a *persistent* property in that it appears to be present from the earliest days of a released software system. In other words, it appears to evolve naturally during development driven perhaps by some deeper principle. This sets the scene for what follows.

- Second, it provides a mechanism which explains the natural appearance of power-law size distributions during development, independently of any representation. The resulting model is a novel modification of the Hartley / Shannon information content within a variational context. This model also attempts to explain observed departures from power-law in smaller components. This phenomenon is demonstrated using major systems.

- Third, it demonstrates that this leads to an approximate linearity between user-defined variable names and component size in executable lines of code. This is demonstrated in real systems.

- Fourth, it then unifies this model with that presented in [10]. It is then able to predict that defects will cluster. This is also illustrated with examples from real systems.

2 Power-law behaviour

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

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

where k is a constant, which on a log p - log s scale is a straight line with negative slope.

Power-law behaviour has been studied in a very wide variety of environments, see for example [21] (economic systems) and the excellent review by [19]. In software systems there has been significant activity, much of it recent, [5], [18], [17], [3], [7], [20], [1], [6] and [10] all discuss power-law behaviour but in rather different contexts. Of these, both Mitzenmacher [17], and Gorshenev and Pis'mak [7], are particularly relevant to the present work.

Mitzenmacher considers the distributions of file sizes in general filing systems and observed that such file sizes were typically distributed with a log-normal body and a Pareto (i.e. power-law) tail. This does not however conflict with the present work as it also leads to a Pareto tail. Perhaps more importantly, his work dealt with sizes in general file systems rather than the symbols of distinct software packages containing numerous strongly related files, together providing the functionality of the software package. This distinction may also be relevant but has not been pursued further here.

Gorshenev and Pis'mak studied the version control records of a number of open source systems with particular reference to the number of lines added and deleted at each revision cycle. The relevance of this will be discussed shortly.

Given the observed appearance of power-law behaviour in such component sizes reported in numerous software packages of very different provenance by [10], and its central part in the theoretical development both there and here, it is of particular interest to investigate why and when such behaviour would emerge, given its link with defect behaviour within components.

Newman in his comprehensive review of power-law behaviour [19], describes a number of potential generating mechanisms.

- Inverses of quantities

- Random walks
- The Yule process
- Phase transitions
- Self-organised criticality
- Combinations of exponentials

Of these, the last named will prove particularly useful in the development which follows.

2.1 Some notes on data presentation and averaging

Before presenting any evidence, it is worth mentioning in passing that software data is typically very noisy for a variety of reasons. Factors of 10 or more between developers in the same group are often noted by researchers in measuring various parameters, for example fault detection capability during inspections, [9]. Furthermore, in comparisons of software components written independently to the same specification even in the same programming language, significant variations in program size measured in lines of code have similarly been reported, [15], [25]. This means that sometimes significant smoothing is necessary to reveal any systematic patterns amongst the inevitable noise. Where necessary in this paper, the arithmetic mean is used without further massaging and its use will be pointed out as appropriate.

With regard to the number of case studies presented in support of each of the testable predictions made, the case studies were chosen at random and (with one exception), at least four very different datasets were used for each as a compromise between shortage of space and statistical reasonableness. For the central result of the paper, six were used.

The systems analysed include large systems worked on by many programmers over a long period, (including the X11R5 library and server, NAG Fortran and C libraries), medium-sized systems worked on for timescales up to 20 years by teams of up to 10, (commercial embedded system, data visualisation, GNU assembler) and many versions of medium-sized systems worked on by one or two developers, (Tcl-Tk GUIs and parsing systems), on timescales up to 10 years.

2.2 Persistence of power-law behaviour in size distributions

In [10], amongst other things, strong evidence of power-law behaviour in the tail of component size distributions for 21 systems of very different size, provenance and programming language is presented. A natural question to ask wherever this information is available, is if this behaviour is persistent from the time of the first release, or whether the power-law behaviour emerged subsequent to its first release as the system matured. If power-law behaviour is present from the earliest days of a released system presumably as a result of some underlying principle acting during development, then the mathematical relationship in [10] predicts that this will act as a driver such that defect growth in a component proportional to xlogx is most likely to take place as the system matures, where x is the number of XLOC. Indeed, some evidence for this functional defect growth was presented using mature systems data but the question remains open whether the power-law behaviour is indeed persistent.

Practical considerations suggest that it would be unusual to expect major changes in component size distribution as a system ages on the general grounds that engineers are reluctant to change working systems too much even as they adapt to changing requirements and other normal maintenance activities. By good fortune, several of the 21 systems analysed in [10] had source revision history, two from the very first release of 7-8 year life-cycles and one from around half way into its 25 year life-cycle from first release. These three systems are highly disparate but to broaden the relevance, an early and a late version of the data visualisation program xgobi / ggobi ([24]), were also analysed.

To reduce the noise in showing the component-size distributions for each release, the data are displayed using the cumulative distribution method described by [19]. Here the shape of the distribution is irrelevant, it is the *change* in shape across revisions which is of particular note. If this does not change substantially, then power-law behaviour in the tail is persistent, since in each case, it was present in the latest releases.

Relevant parameters of these four packages are shown as Table 1.

Package	Language	releases	Years	start XLOC	end XLOC
Numerical library	Fortran	8	12	90,198	266,123
Geophysical modelling	Tcl-Tk	44	7	6,227	11,078
Language parser	C	27	8	35,851	65,270
Visualisation	C	2 shown	17	$\sim 12,000$	$\sim 48,000$

Table 1: Packages used to show power-law persistence

Figure 1 shows the component size distributions for each official release of a widely used numerical library (the NAG Fortran library) from release 12 through release 19, spanning around 12 years. The last release analysed, release 19, comprised 3659 components containing altogether almost 270,000 XLOC. Even though the library almost trebled in size over this period, there is little substantial change in the component size distribution across this time period. For general interest, the data is shown for all component sizes.

It remains possible that substantial change might have taken place in the releases prior to release 12. Although the data were not available to confirm this, it would be most unlikely for a scientific subroutine library to change significantly over its life-cycle by the very nature of its functionality. The solutions of mathematical algorithms hardly vary once implemented.

Figure 2 shows the development of a graphical user interface of approximately 11,000 lines of Tcl-Tk code used in geophysical modelling across all 44 revisions from its first appearance in 2002 to the present day, (only every 4th version is shown for clarity). Again it can be seen that there is no substantial difference in component size distribution across the entire released life-cycle even though the system effectively doubles in size. The power-law behaviour in the tail was already present at first release, it is not an emergent property.

A third system is shown as Figure 3. Again this is in a different language, (this time C), a totally different application area (high-integrity C parsing tool) and is of considerable size, (in this case around 65,000 source lines). This system spans 27 separate releases across an 8 year life-cycle and in this figure every 3rd release is shown for clarity. Again, no substantial change is observed and again it must be concluded that the power-law behaviour in the tail is persistent.

The fourth system is shown as Figure 4. Although again in C, this is the well-known visualisation system xgobi / ggobi, [24] and again no evidence of substantial change in component size distribution is evident across the 17 years separating these two versions during which time the system increased in size by a factor of four.

Given the very disparate nature of these four systems, it can be tentatively concluded that component size distributions do not appear to change substantially across long life-cycles of medium to large packages, (here the range is 11,000 - 270,000 XLOC in three different languages and very different application areas). In other words, *the power-law behaviour in the tails of component sizes is a persistent property, present at first release and it does not emerge during the maintenance cycle, even when that doubles, triples or even quadruples the initial released system size as is the case here.*

This paper will now attempt to answer why such behaviour appears to be persistent. Gorshenev and Pis'mak [7] address a related problem, that of explaining why the added and deleted code in an evolving system should obey a power-law distribution. To this end, they use a model of software evolution based on natural selection. Here, a somewhat different problem is being addressed, that of explaining why power-law behaviour in the tail appears to be present from a system's first release. To this end, an entirely different approach will be taken based around information content.

3 A mechanism for persistence

To be present in the first release of such disparate systems, it is obvious that any such power-law behaviour must evolve naturally as the functionality of a system is implemented in a software context during development and must therefore be intimately related to that functionality in some sense and this will be pursued here.

Historically, functionality has proven to be an elusive goal to quantify for computer scientists as evidenced by the fact that the number of lines of code still appears to be the dominant measure of *size*, and by common association, functionality, 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:-

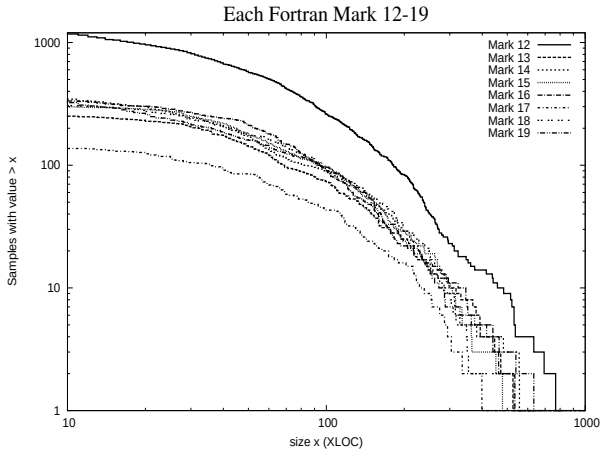

Figure 1: The distribution of component sizes displayed various versions of a major Fortran numerical library as reported by [11]. This data covers the development from about a half of the way into the life-cycle up to the present day. No substantial change in distribution is evident across this time-period.

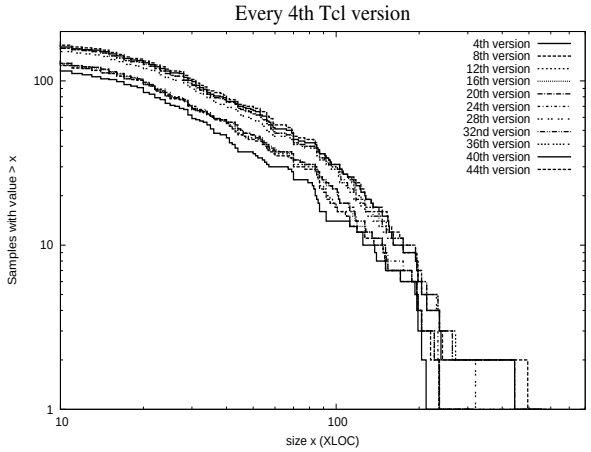

Figure 2: The distribution of component sizes for every 4th release in the first 44 versions of a GUI development written in Tcl-Tk and covering some 7 years of development. Across the entire life-cycle, there is no substantial change in component size distribution.

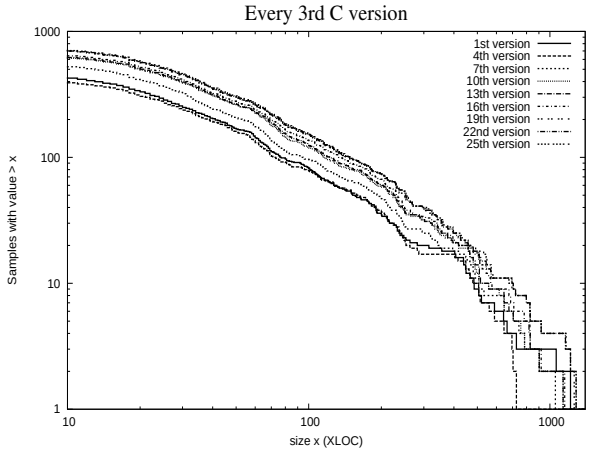

Figure 3: The distribution of component sizes displayed for every 3rd release in the first 27 versions of a high-integrity parsing engine written in C and covering some 8 years of development. Across the entire life-cycle, there is no substantial change in component size distribution.

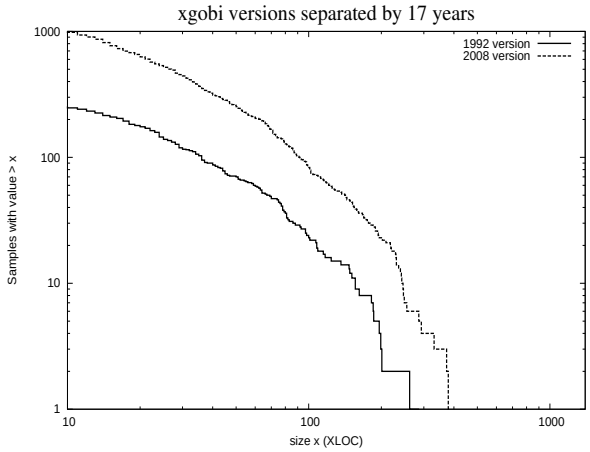

Figure 4: The distribution of component sizes displayed for two releases of the xgobi/ggobi visualisation engine separated by 17 years. Across the entire life-cycle, there is no substantial change in component size distribution.

- 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 very highly correlated with normalised correlation coefficients typically > 0.9 in the C populations used here.

The relationship of any of the line of code measures 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 significant difficulties as shown by [13] and [14].

To attempt to circumvent these problems, a more basic approach based on information theory will be taken here.

3.1 Hartley-Shannon information content

Now it will be recalled from the initial discussion of power-law behaviour that Newman [19] gives a list of possible mechanisms for the evolution of power-law behaviour of which combination of exponentials turns out to be a fruitful avenue to pursue for software systems as will now be seen.

In essence if some quantity y has an exponential probability distribution

$$p(y) \sim e^{ay} \tag{2}$$

and some other quantity of interest x behaves like

$$x \sim e^{by} \tag{3}$$

Then the probability distribution of p(x) is given by

$$p(x) \sim x^{\frac{a}{b}-1} \tag{4}$$

which is power-law behaviour. The potential attraction of this for software systems is that it has been used before in a textual context by Miller [16] who applied it in order to attempt to explain the observed power-law distribution of the frequencies of words in texts. Miller's work was criticised in that it assumed randomly typing on a keyboard to generate valid words which of course, is far from the case in a coherent text. This led Hartley [8] to formulate ideas based on information content which were then developed into a theory of information transmission by Shannon [22] as described in Cherry's unifying book, [4].

3.2 The information content of a component

Hartley [8] 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.

In the context of a programming language, a sign is a symbol of the language potentially containing multiple characters and is sometimes called a *token*, (tokenisation is also known as lexical analysis and is the first stage of language compilation whereby individual characters are glued together into the higher-level objects in which the syntax of the language is defined).

Tokens occur in two forms. The first form is fixed in the sense that these tokens are defined by the programming language definition itself. Examples include the characters **{**, and **}** which delineate blocks in C and C++ and also keywords such as **if**, **while**, **continue** and so on. The programmer has no choice with these other than to use them or not.

The second form of token is defined by the programmer and is arbitrary apart from certain lexical constraints such as maximum number of characters, character content, beginning character, (and in some languages, avoidance of reserved names). These are the *identifiers* or *variable names* used in a program and are essentially addresses of storage locations which the programmer uses as working space in the implementation of algorithms. These are supplemented by *constants* also defined by the programmer, such as the value of π for example.

It should be noted that extraction of such tokens from a program is not a trivial process and requires developing tools which mimic the front-end of compilers. The real systems studied in this paper are all in C, Fortran or Tcl-Tk as such front-ends had already been developed by the author for previous projects. As an example of some of the subtleties that can arise, C is a reserved keyword language, meaning that tokens such as **if** can only be used as the pre-amble of a conditional statement. In Fortran 77 however, keywords are not reserved and **if** can either be used as the pre-amble of a conditional statement or as a user-defined variable name depending on its context. The language front-ends have to be good enough to recognise such subtleties to produce valid data. In this case, both the Fortran and the C front-ends had been tested against the relevant formal validation suites to build confidence in parsing and therefore in the process of token extraction.

The relationship of programming language tokens with lines of code will be returned to later when real systems are examined.

Suppose the i^{th} component of a software system has t_i tokens in all, constructed from an alphabet of a_i tokens. Following the discussion above,

$$a_i = a_f + a_v(i) \tag{5}$$

where a_f is the alphabet of fixed tokens and $a_v(i)$ is the alphabet of programmer-defined tokens and is clearly dependent on i, since programmers are free to create them as and when desired. In contrast, assuming a_f is fixed for all components is not a bad approximation as even the simplest programs can consume many of the commonly occurring fixed tokens of a language as a form of syntactic overhead, (for example, a commonly occurring implementation of a bubblesort program of only 8 lines uses 20 fixed tokens but only 6 user-defined variable names). Furthermore, a_f is usually significantly less than the total number of available tokens in a programming language anyway as numerous tokens are very rarely used, for example, the 10 *trigraphs* or the *goto* in ISO C. For these reasons, a_f can be considered as the *effective* size of the fixed alphabet.

The number of ways of arranging the tokens of this alphabet in component i is therefore $a_i^{t_i}$. Following Hartley, the quantity of information in component i will therefore be defined as

$$I_i = log(a_i)^{t_i} = t_i log a_i \tag{6}$$

3.3 The most likely distribution of components

Using equation (6), an abstract model of information distribution will now be derived based on a refinement of the development used in the previous paper, [10].

Suppose that a system is made up of M components each of size t_i tokens such that the total size T is given by

$$T = \sum_{i=1}^{M} t_i \tag{7}$$

Then the number of ways of organising this system is given by:-

$$W = \frac{T!}{t_1!t_2!..t_M!} \tag{8}$$

Also suppose there is some externally imposed entity ε_i associated with each token of component i whose total amount is given by

$$U = \sum_{i=1}^{M} t_i\varepsilon_i \tag{9}$$

Using the method of Lagrangian multipliers as described in [10], the most likely distribution satisfying equation (8) subject to the constraints in equations (7) and (9) will be found. This is equivalent to maximising the following variational

$$logW = TlogT - \sum_{i=1}^{M} t_i log(t_i) + \lambda\{T - \sum_{i=1}^{M} t_i\} + \beta\{U - \sum_{i=1}^{M} t_i\varepsilon_i\} \tag{10}$$

where λ and β are the multipliers. Setting $\delta(logW) = 0$ and using the assumption that T and the $t_i \gg 1$ leads to

$$0 = -\sum_{i=1}^{M} \delta t_i\{log(t_i) + \alpha + \beta\varepsilon_i\} \tag{11}$$

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

$$log(t_i) = -\alpha - \beta\varepsilon_i \tag{12}$$

Using equation (7) to replace α, this can be manipulated into the most likely, i.e. the equilibrium distribution

$$t_i = \frac{Te^{-\beta\varepsilon_i}}{\sum_{i=1}^{M} e^{-\beta\varepsilon_i}} \tag{13}$$

To see the relevance of this to the evolution of software systems during development, consider this equilibriation in terms of the mental processes in developers as they re-arrange tokens gradually into programs of the desired functionality. This process of mental translation from the specification domain into the programming domain by both small (evolutionary) steps and large (revolutionary) steps is not well understood but some intriguing insights can be gained into it from quotations like the following made on p.200 of [12] about the evolution of the *diff* file comparison program:

> "I had tried at least three completely different algorithms before the final one. **diff** is a quintessential case of not settling for mere competency in a program but revising it until it was right - M.D. McIlroy, 1976"

Every developer will identify with this process. Matching and optimising designs by the continuous re-organisation of tokens of a programming language, 'until it is right', goes to the very heart of programming. Simulating it using a variational process which proves of great value in understanding the evolution of complex systems in the natural world therefore seems entirely natural.

Following [21] and defining $p_i = \frac{t_i}{T}$ and referring to equation (9), p_i can be interpreted as the probability that a component is found with a share of U equal to ε_i. Manipulating equation (13) then yields

$$p_i = \frac{e^{-\beta\varepsilon_i}}{\sum_{i=1}^{M} e^{-\beta\varepsilon_i}} \tag{14}$$

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 t_i tokens actually occurs.

So far this is a similar development to that followed in [21] and [10] for example, although it generalises the argument by using tokens of programming languages, which are the natural currency of information theory.

3.4 Merging with information theory

To blend these two developments, the same computational device used in [10] will now be applied again. Using equation (6), introduce the total amount of

information I as the sum of the information in each component as follows:-

$$I = \sum_{i=1}^{M} t_i(\frac{I_i}{t_i}) \tag{15}$$

This leads directly to the identification of ε_i with $(\frac{I_i}{t_i})$ in equation (13). In other words, each token of component i has an information density associated with it given by $(\frac{I_i}{t_i})$. This assumes that the information per token in a single component is constant but that this can vary amongst components. This seems entirely reasonable in that it suggests that no particular token is any more important than any other when developing a particular component as some functional entity, however it allows for the fact that this can vary amongst different components which fits well with intuition that some functional entities are in some sense 'harder' than others. Note finally that introducing this additional functional dependence of ε_i on t_i does not disrupt the development which led to equation (14) as ε_i is fixed externally by assumption.

Equation (14) can then be written as

$$p_i = \frac{e^{-\beta \frac{I_i}{t_i}}}{Q(\beta)} \tag{16}$$

where

$$Q(\beta) = \sum_{i=1}^{M} e^{-\beta \frac{I_i}{t_i}} \tag{17}$$

Combining equations (16) and (6) then gives

$$p_i = \frac{e^{-\beta log a_i}}{Q(\beta)} \tag{18}$$

This of course is power-law behaviour

$$p_i = \frac{(a_i)^{-\beta}}{Q(\beta)} \tag{19}$$

This states in essence that subject to the constraints that the total number of tokens T and the total amount of information I in a system is conserved, and that the information density per token is constant within a particular component, then the most likely distribution of component sizes to evolve will obey a probability distribution based on a power-law in the total alphabet used to construct each component.

Given its central position in what follows, it is useful to retrace the assumptions. These are

1. The variational method assumes that both t_i and T are $\gg 1$. This turns out to be a very good approximation for nearly all the data here. Components with $t_i < 20$ are rare. (Note that this is therefore more accurate than the method used in [10] which used executable lines of code as an indivisible quantity.)

2. The variational method assumes that T is kept constant whilst the most likely solution is found. It should be noted that this is not its actual size at any point in time, but the eventual size defined by its intended functionality in an ergodic sense. In other words, if the same system was produced many times independently, then for a particular T, the variational method finds the most likely distribution of t_i subject to the constraints.

3. The variational method assumes that the total information content I is kept constant. I is not the same as functionality however and this distinction will be discussed shortly.

4. The information content per token is assumed fixed for a component but may vary between components. It is easy to relate to the idea that some components are more difficult to implement than others. Whether information density per token is constant is moot but an average information density per token for a component is a reasonable compromise.

It is useful to re-iterate that the development is completely independent of any programming language or paradigm. It simply assumes the existence of an alphabet of tokens which can be combined in some way to represent information.

3.4.1 Small components

Using simple properties of the relative sizes of the fixed and variable alphabets a_f and $a_v(i)$ respectively, the shape of this curve on a log-log scale can be anticipated. This is slightly complicated by the fact that the approximation inherent in the development of equation (14) is less good for smaller t_i but as was noted above, even for very simple programs $t_i \gtrsim 20$, so the approximation holds up well enough to get a good idea of the behaviour for smaller components as follows:-

Combining equations (5) and (19) gives

$$p_i = \frac{(a_f + a_v(i))^{-\beta}}{Q(\beta)} \tag{20}$$

For small components, as has been seen, it is reasonable to assume that the number of fixed tokens will tend to dominate the total number of tokens. In other words, $a_f \gg a_v(i)$. Equation (20) can then be written

$$p_i = (a_f)^{-\beta} \frac{(1 + \frac{a_v(i)}{a_f})^{-\beta}}{Q(\beta)} \tag{21}$$

In other words,

$$p_i \sim (a_f)^{-\beta} \tag{22}$$

which implies that p_i will be tend to a constant for small components on a log-log plot. This then leads to

3.4.2 Testable prediction 1

The probability p_i of a component appearing with t_i tokens for small components should tend to a constant on a log-log plot. For large components, it should obey a power-law in the total alphabet a_i, giving linear behaviour with a negative slope on a log-log plot.

This fundamental result was tested using six randomly selected systems of very different sizes[1], and of very different application area for increased confidence. In this context, random means that the packages were selected to appear in the diagram before they were analysed. The systems chosen were the NAG scientific subroutine library (Fortran), a commercial embedded communication system (C), the X11R5 library (C), the X11R5 server (C), the data visualisation package xgobi / ggobi (C) and an ancient version of the GNU assembler GAS (C), version 1.37. Figure 5 shows the component size distribution for each, plotted against their respective total alphabets a_i using the cumulative distribution method of [19]. The behaviour predicted by equations (19) and (22) appears to be present in each case[2] and the larger the package, the more pronounced the behaviour, which is consistent with the large ensemble statistical mechanical argument used.

[1]The largest has over 3500 components and the smallest has just 35.

[2]For a power-law probability distribution function (pdf) such as given by equation (19), the cumulative distribution function (cdf) has the same functional behaviour but with a different slope, [19], so the above arguments hold equally well for the cdf.

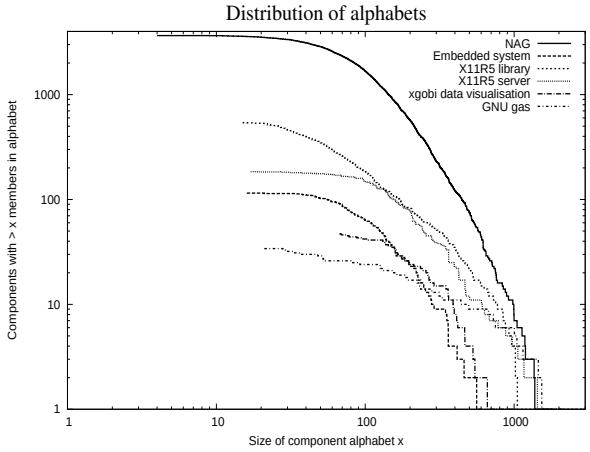

Figure 5: The size distribution of the alphabets a_i for each component of the six independent systems described in the text. In each case, agreement is excellent with the tail exhibiting power-law behaviour predicted by equation (19) tending to a constant for smaller components as predicted by equation (22).

This is the central result of this paper but it can be used to predict other patterns as will now be seen.

4 Approximate linearity of alphabet size in large components

For large components, it is reasonable to assume that the number of user-defined tokens will at some point dominate the total number of tokens, since the other tokens are fixed by the language. In other words, $a_v \gg a_f$. Now it has already been seen above in equation (19) strongly supported by the examples of Figure 5 that strong power-law behaviour is evident in the tail of component size with respect to the total size of the alphabet. However, the cumulative distribution function method used in Figure 5 is equivalent to rank/frequency plots, (c.f. Appendix A of [19]). Furthermore, such linearity means that $p_i \propto \frac{1}{r}$, where r is the rank ordering. This is Zipf's law and implies, following [23] p. 163, that the following approximate relationship can be derived for the total number of tokens in a component t_i given the distinct alphabet a_i used to construct it

$$t_i \approx a_i(\gamma + ...) \tag{23}$$

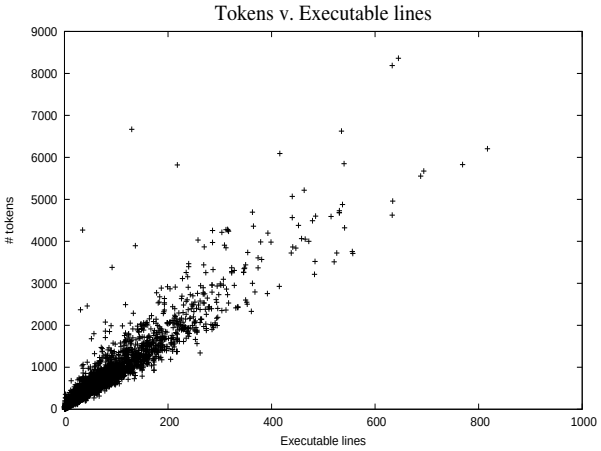

Figure 6: A scatter plot of total number of tokens against XLOC for the NAG Fortran library.

where γ is Euler's constant and only the first term in the expansion has been retained, (the second term is logarithmic in a_i).

Using equation (5), equation (23) can be written as

$$t_i \approx \gamma(a_f + a_v(i)) \approx \gamma a_v(i)(1 + \frac{a_f}{a_v(i)}) \sim a_v(i) \tag{24}$$

In other words, the total number of tokens will be approximately linear in the number of user-defined variable names in the tail of the distribution. Given also that the total number of tokens is approximately linear with XLOC as exemplified by Figure 6 for the NAG Fortran library, (the same is observed for the other studies shown here). This leads to[3]

4.0.3 Testable prediction 2:

The number of user-defined variable names will be approximately linear with the size of a component in XLOC.

Figure 7 shows the number of user-defined variable names plotted against size of component in XLOC for the NAG Fortran library. Here user-defined variable names have been collected together in bins of 5 and plotted against the mean executable line count for that bin to reduce noise in the data. In

[3]The reason for translating to XLOC is to allow this result to be more widely testable. To extract the tokens directly, access to sophisticated parsers for each programming language is necessary, as described in the text.

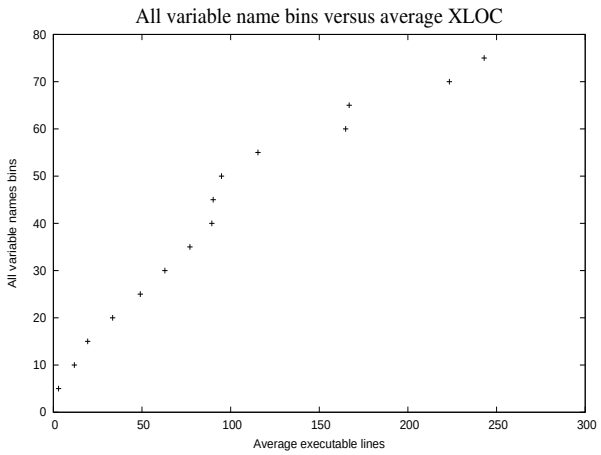

Figure 7: An approximately linear relationship between the number of user-defined variable names and component size in XLOC for the NAG Fortran library, supporting *Testable prediction 2*.

the case of C systems, it is a little less clear how to define variable names and components as in C, the file and the function both act as interfaces in different ways, (functions in the standard way, but files through file scope when several functions share the same file as is often the case in the systems studied here). Even so, approximate linearity for files is again found as shown in Figures 8 (the X11R5 library) and 9 (an embedded control system). The same is true of the X11R5 server, although this is not shown.

No attempt will be made here to assess the statistical significance of these as they follow by making an empirical observation about the distribution of tokens and XLOC on the back of the central result of equation (19). It is merely suggestive of a general trend.

5 Unification with defect models and clustering

In a previous paper, [10], defect growth of xlogx where x is the number of XLOC, was shown to be intimately related with power-law component size distributions in very disparate systems. There is another very interesting empirically observed property of defects in software systems; they cluster. For example, the effect is sufficiently pronounced that [2] include it as property number four in a defect top ten list and quote several sources noting that

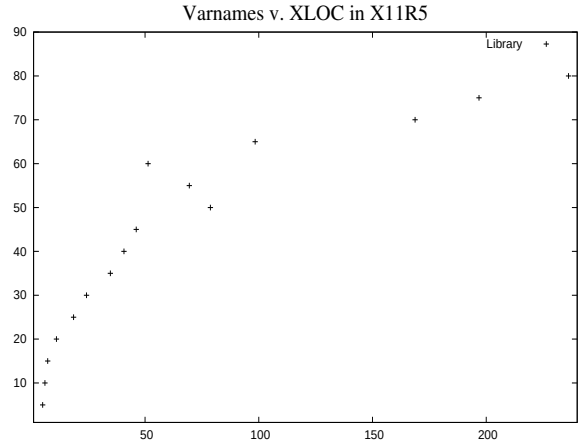

Figure 8: An approximately linear relationship between the number of user-defined variable names and component size in XLOC in the X11R5 library, supporting *Testable prediction 2*.

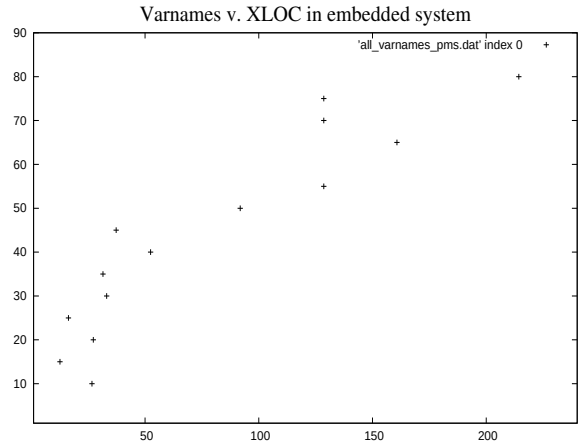

Figure 9: An approximately linear relationship between the number of user-defined variable names and component size in XLOC in an embedded control system, supporting *Testable prediction 2*.

"80% of the defects appear in 20%" of the modules. This is however a well-known property of Pareto or power-law distributions.

The model presented here gives a possible explanation for this and moreover, suggests that they may actually be different facets of the same phenomenon. Consider equation (19). Supposing that each time a token t_i is used, the assumption is made that there is some constant probability p of making a mistake, i.e. creating a defect. Then, the total number of defects in a component, $d_i = p.t_i$. Substituting this in equation (19) and using equation (23) again, yields

$$p_i = \frac{(a_i)^{-\beta}}{Q(\beta)} \sim (t_i)^{-\beta} \sim (d_i)^{-\beta} \tag{25}$$

leading to,

5.0.4 Testable prediction 3:

The probability that d_i defects will appear in a component approximately obeys a power law or Pareto distribution if the probability of making a mistake is constant with any token.

Quite apart from the studies reported by [2], this can be demonstrated here on two datasets for which there is a particularly good defect record of considerable maturity, that of the NAG scientific subroutine Fortran and C libraries as shown in Tables 2 and 3, [11]. In both cases, defect clustering is manifestly present.

Number of defects	Number of components	NXLOC
0	2865 (78.30%)	179947 (67.62%)
1	530 (14.48%)	47669 (17.91%)
2	129 (3.53%)	14963 (5.62%)
3	82 (2.24%)	13220 (4.97%)
4	31 (0.84%)	5084 (1.91%)
5	10 (0.27%)	1195 (0.45%)
6	4 (0.11%)	1153 (0.43%)
7	3 (0.08%)	1025 (0.39%)
8	0 (0.00%)	0 (0.00%)
9	0 (0.00%)	0 (0.00%)
10	4 (0.12%)	1653 (0.62%)
11	1 (0.03%)	214 (0.08%)

Table 2: Defect counts v. subroutines and executable lines of code in the NAG Fortran library.

Number of defects	Number of components	NXLOC
0	1506 (66.43%)	194250 (52.47%)
1	538 (23.73%)	112572 (30.40%)
2	166 (7.32%)	40935 (11.06%)
3	46 (2.03%)	17649 (4.77%)
4	9 (0.40%)	4317 (1.17%)
5	2 (0.09%)	490 (0.13%)

Table 3: Defect counts v. functions and executable lines of code in the NAG C library.

6 Threats

The main threat to this paper is as always, the amount of evidence presented. As stated at the beginning, it was felt that at least four very different systems for each of the testable hypotheses with the single exception stated, gave a reasonable balance between space and statistical reasonableness but there remains the possibility they are not sufficiently representative.

In some cases the data have been averaged within bins. In each case, the mean was used as the amount of data was sufficient but as pointed out, software data is by its very nature, noisy.

With regard to the argument leading up to a prediction of power-law behaviour of defects represented by equation (25), the description of defects in terms of a constant probability of making a mistake with a particular token is almost certainly too simplistic although it may be a reasonable model on average.

7 Information content and functionality

Finally, this paper would not be complete without discussing the relationship between information content and functionality. First of all, it is worth noting that it is natural to over-emphasize the relationship between information content and meaning, and by inference, functionality. Indeed Cherry [4], p. 50, specifically cautions against this noting that the concept of information

based on alphabets as extended by Shannon and Wiener amongst others, *only relates to the symbols themselves* and not their *meaning*. Indeed, Hartley in his original work, defined *information* as the successive selection of signs, rejecting all meaning as a mere subjective factor.

In other words, the development using information content from equation (6) onwards leading to the relationship expressed by equation (19) is fundamental but it says little if anything about functionality. *This strongly suggests that power-law behaviour is nothing to do with functionality and is related simply to the use of alphabets of tokens to build texts such as programming systems.* In other words, it operates at an even lower level than functionality or meaning.

The proper study of meaning is known as *semiotics*. In this discipline, rules acting on signs or tokens are split into three categories:-

- Syntactic rules (rules of syntax; relations between signs)
- Semantic rules (relations between signs and the things, actions, relationships and qualities known collectively as *designata*)
- Pragmatic rules (relations between signs and their users)

The development described here relates only to the first category.

Interestingly however, the existence of power-law behaviour in defect distributions suggests that defects may have more in common with errors in the use of the tokens of a language in the simple manner described earlier, than in their meaning.

8 Conclusions

The paper presents four contributions each supported by real systems data of different provenance and programming language.

- Power-law component size behaviour appears to be persistent throughout the development life-cycle. In the systems analysed, no essential changes were noted over long release life-cycles. This is predicted by the next contribution.

- Using variational principles suggested in [10] merged with arguments from Hartley-Shannon information theory, it is predicted that the probability p_i of a component appearing with t_i tokens in any software system, whatever its implementation details, obeys a power-law with respect to the size of its distinct alphabet of tokens a_i,

$$p_i \sim (a_i)^{-\beta} \tag{26}$$

- Using the simple law of equation (26) and the observation that the number of tokens per XLOC is approximately constant, it is further predicted that the number of user-defined variable names must be approximately linear with component size in XLOC.

- Also using equation (26) along with the simple assumption that there is a constant probability of making a mistake on any token, it is also predicted that defects will cluster as a power-law distribution leading directly to the widely-observed Pareto or 80:20 rule.

In addition, it provides a unified model with [10], whereby during the development phase, subject to constraints on the total information content and the total size, a power-law distribution in the alphabet used to construct each component will emerge. This in turn implies the approximate power-law behaviour of component size with respect to XLOC observed in [10].

When such a system is released, subject to the constraints that size and total number of defects is fixed, those defects will then appear roughly as xlogx where x is the number of XLOC and that this phenomenon also manifests itself as clustering.

Finally, it is hypothesized that,

- The equilibriation process which takes place leading to equation (19) takes place in the programmers' minds as functionality is gradually translated into code.

- Power-law behaviour is more fundamental than functionality and appears to be related only to the alphabet of tokens used.

- Defect is intimately related to information content. This can be seen here in the close relationship between the variational principle used here in the development phase in which the externally specified variational constraint $\varepsilon_i = (\frac{I_i}{t_i})$ and that used by [10] in the release phase where

$\varepsilon_i = (\frac{d_i}{n_i})$, recalling that t_i the number of tokens and n_i, the number of XLOC, differ on average only by a scale factor as described earlier. d_i is the number of defects in component i. This may indicate that defect is more closely related to the use of the tokens themselves rather than the meaning of the tokens.

These hypotheses will be examined in a follow-up work as space prohibits a more detailed discussion here.

References

[1] G. Baxter, M. Frean, J. Noble, M. Rickerby, H. Smith, M. Visser, H. Melton, and E. Tempero. Understanding the shape of java software. *OOPSLA '06*, 2006. http://doi.acm.org/10.1145/1167473.1167507.

[2] B. Boehm and V.R. Basili. Software defect reduction top 10 list. *IEEE Computer*, 34(1):135–137, 2001.

[3] D. Challet and A. Lombardoni. Bug propagation and debugging in asymmetric software structures. *Physical Review E*, 70(046109), 2004.

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

[5] D. Clark and C. Green. An empirical study of list structures in lisp. *Communications of the ACM*, 20(2):78–87, 1977.

[6] G. Concas, M. Marchesi, S. Pinna, and N.Serra. Power-laws in a large object-oriented software system. *IEEE Trans. Software Eng.*, 33(10):687–708, 2007.

[7] A.A. Gorshenev and Yu. M. Pis'mak. Punctuated equilibrium in software evolution. *Physical Review E*, 70:067103–1,4, 2004.

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

[9] L. Hatton. Testing the value of checklists in code inspections. *IEEE Software*, 25(4):82–88, 2008.

[10] L. Hatton. Power-law distributions of component sizes in general software systems. *IEEE Transactions on Software Engineering*, July/August 2009.

[11] 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.

[12] B.W. Kernighan and R. Pike. *The Unix programming environment.* Prentice-Hall software series, 1984. ISBN 0-13-937681-X.

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

[14] 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.

[15] J.C. Knight and N.G. Leveson. An experimental evaluation of the assumption of independence in multi-version programming. *IEEE Transactions on Software Engineering*, 12(1):96–109, 1986.

[16] G.A. Miller. Some effects of intermittent silence. *American Journal of Psychology*, 70:311–314, 1957.

[17] Michael Mitzenmacher. Dynamic models for file sizes and double pareto distributions. *Internet Mathematics*, 1(3):305–333, 2003.

[18] Christopher R. Myers. Software systems as complex networks: Structure, function and evolvability of software collaboration graphs. *Physical Review E*, 68(046116), 2003.

[19] M. E. J. Newman. Power laws, pareto distributions and zipf's law. *Contemporary Physics*, 46:323–351, 2006.

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

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

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

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

[24] D.F. Swayne, D. Cook, and A. Buja. Xgobi: A system for multivariate analysis, 1998. http://www2.research.att.com/areas/stat/xgobi/ and http://www.ggobi.org/, accessed 12-Dec-2009.

[25] Meine J.P. van der Meulen. The effectiveness of software diversity. Ph.D. Thesis, City University, London, 2008.