Measure Software – and its Evolution – Using Information Content

Tom Arbuckle
Electronic and Computer Engineering
University of Limerick
Ireland
tom.arbuckle@ieee.org

ABSTRACT

To be able to examine software evolution – variation in software over a sequence of releases – or to compare differing versions of software with each other, we need to be able to measure artefacts representative of the software or its creation process. One can find in the literature a multitude of approaches to both measuring software – by defining and applying software metrics – and to examining software evolution in terms of these metrics. In this position paper, we claim that information content, specifically the (relative) Kolmogorov complexity, is the correct and fundamental tool for the measurement of software artefacts. Experimental results obtained from an analysis of the project *udev* demonstrate utility: future work should explore the breadth of applicability and determine the full scope of the approach.

Categories and Subject Descriptors

D.2.8 [**Software Engineering**]: Metrics—*complexity measures, performance measures*; D.2.7 [**Software Engineering**]: Distribution, Maintenance, Enhancement; H.1.1 [**Models and Principles**]: Systems and Information Theory

General Terms

Design, Documentation, Measurement, Theory

Keywords

Kolmogorov complexity, software measurement, software evolution, information theory, similarity metric, CompLearn

1. INTRODUCTION

This paper's thesis is that, fundamentally, software artefacts of all types can best be measured and compared in terms of their information content as measured by their Kolmogorov complexity.

If we wish to measure or investigate how software evolves – how subsequent releases of a software project differ from each other – we need to have some means of measurement by which we can reveal the course of the evolution.

In software engineering (hereafter SE) we measure software using software metrics. Software metrics are (abstract) tools that we apply to artefacts representative of the software or of its production process to produce numerical representations. In turn, we hope that these numbers represent what we do when we create software and thereby provide us with the information we need to measure and predict.

Here we take the word 'artefact' to mean any product or by-product of the software creation process that we can then use as input to one of our measurement tools. In studying software evolution, we apply metrics to the artefacts taken at given points in time or corresponding to different, generally sequential, releases of the software project. In addition to metrics that we can apply to single data sets (releases) we can, alternatively, look for metrics that explicitly involve either the process of creation or distinctions between these different data sets.

There are very many software metrics [18, 27, 15, 10, 20, 35, 14]. There are many artefacts that have been considered for the measurement of software evolution [21].

We have two main reasons for claiming that Kolmogorov complexity is the 'best' measurement device for software artefacts. Firstly, the Kolmogorov complexity is a fundamental mathematical concept which is, by definition, a measurement of the number of bits of information intrinsic to an object. Being fundamental and having a mathematical definition, it is unlikely that there will be significant disagreement about its interpretation (of and by itself). Secondly, its fundamental nature means that it involves no tunable parameters, no expert evaluations, no calibration. Therefore it can and should serve as a basis and as an underpinning for other derived metrics extant in the literature that have already been shown to have practical worth and, in so doing, should provide an unequivocal means of comparison.

There are other reasons behind our claim – and we are also aware of some criticisms of it. These will be discussed in the following sections. We will continue to expound on our claim, showing its relation to previous work, in the next section. In the third section of this paper, we will provide some evidence of the validity of the claim by showing how the (relative) Kolmogorov complexity can be employed in a short proof-of-concept example. We study the open source project *udev* [24]. In our conclusion, as well as summing up, we will briefly discuss some further potential opportunities for application of the method.

IWPSE-Evol'09, August 24–25, 2009, Amsterdam, The Netherlands.

2. EXPOSITION

We claim that a theoretical basis for the measurement of software will come from the field of information theory, that it will be based on Kolmogorov complexity, and that relative measures of Kolmogorov complexity are already available and of practical significance. The reasoning behind these claims is now dealt with in detail.

2.1 Background

We can define any software metric that seems useful to us – and many people have done so. When considering software evolution, the class of potential metrics is broader still. Practitioners have derived some utility from at least some of the proposed software metrics. Through *ad hoc* rules-of-thumb and heuristics, they have sought to bring some order to the practice of SE by deriving design advice from numbers in some way representative of software. Software, in turn, reflects the decisions made in its design and construction.

Michael Jackson [19] has argued that SE has not yet specialised to the extent that software construction can be considered the purview of 'normal design' as opposed to 'radical design'. Indeed, in cases where software construction can be reduced to normal design (the minor or automated modifications of existing designs) risk is minimised and predictability is higher. Many, if not all, of the software metrics in use see greatest success when used within the 'normal design' paradigm, areas where the metrics are known to be 'good'.

However, despite strong arguments advocating the contrary, it is still the case that much of software development is radical rather than normal. Practitioners' battery of heuristics can, as a result, provide only guidelines and cannot be more finely tuned to software measurement or the revelation of software evolution without losing their generality. If there were some underlying theoretical approach that could justify and validate the measurement of software, it could instead serve both as a basis for future modelling and as a common thread, tying previous indicators together.

Information theory is the branch of science that we have for reasoning about information: if we agree that software is information, then a theoretical basis for measuring and comparing software will come from this field. This assertion is validated by contemporary work, such as that by Harman [16, 17] – describing the need for information theoretic metrics as fitness functions in search-based SE with Lutz [26] as an exemplar – or that by Clark *et al.* [13] or by McCamant and Ernst [28] – on information flow with implications particularly for software security. It was also the conclusion of early workers in SE. van Emden's theoretical work [33, 34] extending themes from Simon and Ando [31] and from Alexander [1] and the practical demonstrations of employing entropy for measurement of software design by Chanon [8, 9] were the first applications of information theory in the new field of SE. These initial forays were followed by many papers seeking to use variants of entropy to measure software or its design or its evolution [35, 22].

Consider the discrete case of Shannon's entropy [30]:

$$H(X) = -\sum_{x \epsilon X} p_x \log p_x. \tag{1}$$

The entropy, $H(X)$, is calculated in terms of the probabilities p_x that the codes x from a set X will occur during their transmission from a source to a sink over a channel. We can ask: in SE terms, what are the symbols here, what do they imply, and how do we measure their probabilities of occurrence? Clearly, given the significant numbers of papers seeking to employ entropy in SE, there are many possible answers. Alternatively, we can try a different approach. Shannon's entropy relates to the significance of occurrence of symbols taken from a known alphabet[1] but says nothing about the information content of the symbols themselves. The information content of the symbols is measured using Kolmogorov complexity.

2.2 Kolmogorov Complexity

Kolmogorov complexity, also known as "algorithmic entropy", was discovered independently by Solomonoff [32], Kolmogorov [23] and Chaitin [7]. The Kolmogorov complexity, $K(x)$, of a binary string x is, by definition, the length of the shortest (prefix-free) binary program to compute x on a universal computer, such as a universal Turing machine. It gives the number of bits to computationally describe x.

Kolmogorov complexity complements entropy. Shannon's concern centres on the characterisation of messages from a random source. Kolmogorov dispenses with probability and considers only individual messages' information content. For the purposes of software measurement, we are claiming that Kolmogorov complexity provides a more appropriate fit.

There is, however, a practical difficulty to this. As a definition based on an abstract universal computing engine (the Turing machine), one might wonder how the value of Kolmogorov complexity can be found. Indeed, as a non-partial recursive function, the Kolmogorov complexity is non-computable. Some form of approximation is needed.

Note firstly, that Kolmogorov complexity and Shannon entropy can be related. It can be shown that the expected Kolmogorov complexity for any distribution will be close to its Shannon entropy. The approximation of the Kolmogorov complexity in terms of entropy is also the device used by Allen *et al.* [2]. They show that

$$E(Len(x)) \geq H(x) \tag{2}$$

where, for an instantaneous code for the domain of x, the expected length per item is $E(Len(x))$. They explain that "the entropy of a distribution of x is the minimum expected length of an instantaneous code for one sample item". Moreover, by using a Shannon-Fano coding of n items from a set X, members x, the Kolmogorov complexity of the set X can then be written ([2], p.189)

$$\widehat{K}(X) = nH(x). \tag{3}$$

By employing graph representations of programs, taking elements from closed sets thus fixing n, they can then use counting arguments to derive $H(x)$ and from that $\widehat{K}(X)$.

One interesting conclusion of their paper is that "Information size is highly correlated with counting size". Given that many SE metrics count features representative of software artefacts - lines, methods, calls - we claim that this result provides some evidence both for our argument but also for those who may claim that existing metrics are good enough. Now we know why.

2.3 Information Distance, Relative Measures

Our interest is in monitoring change over software releases, so we are interested in means of comparison. Kolmogorov

[1] What is the alphabet for radical design?

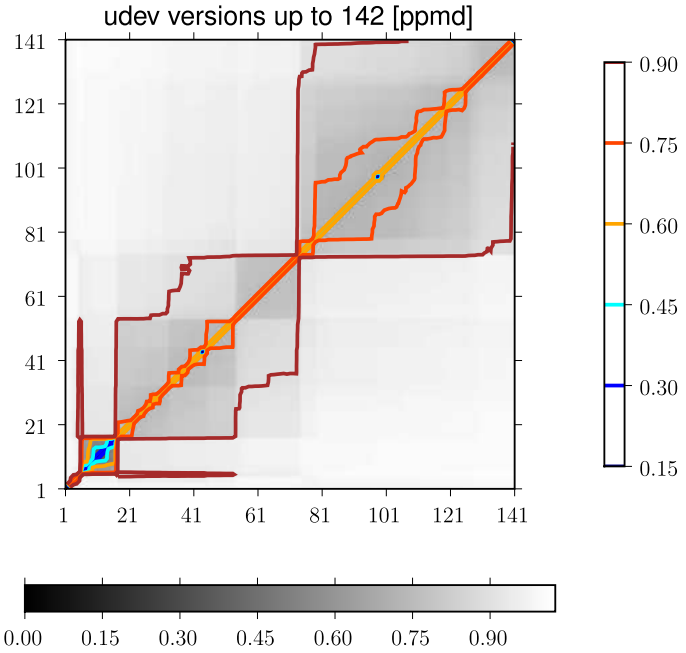

Figure 1: Greyscale NCD distance matrix for all 141 releases of *udev*. Note. The coloured contour lines occlude the results for self-comparisons, all of which have value zero.

complexity measures the amount of information in an object. Bennett *et al.* [5] asked how the information distance between two objects could be appropriately measured. Concurrently with establishing deep connections between this field and the thermodynamics of computation, they showed that an appropriate metric was

$$E_1(x,y) = max(K(x|y), K(y|x)). \tag{4}$$

$K(x|y)$, for example, is the conditional Kolmogorov complexity of x given y, the length of the shortest program for a universal Turing machine to output x for an input y.

Often it is the case that we wish to know difference in a proportional rather than an absolute sense. A difference of 100 bits between two objects of 1 million bits may be unimportant but if the objects consist of 100 bits then the difference will certainly be more noticeable. This was the motivation for the introduction of the normalised information distance, NID, introduced by Li *et al.* [25]. The NID is defined by

$$NID(x,y) = \frac{max\{K(x|y), K(y|x)\}}{max\{K(x), K(y)\}} \tag{5}$$

They showed that the NID satisfies the metric properties up to an additive term of $O(1/K)$, where K is the maximum of Kolmogorov complexities involved. Then we can interpret $1 - NID(x,y)$ as the number of bits of shared information per bit of information of the string with more information.

The mathematics for the NID is exact. The final step we need is that made by Cilibrasi and Vitányi [12]. They showed that the NID could be approximated by using real-world compression mechanisms provided that those compression mechanisms possessed some common properties. The new measure is the normalised compression distance, NCD:

$$NCD(x,y) = \frac{C(xy) - min\{C(x), C(y)\}}{max\{C(x), C(y)\}}. \tag{6}$$

Here xy denotes the concatenation of x and y and $C(x)$, for example, denotes the approximation of a Kolmogorov complexity $K(x)$ by the length of the compressed data produced by an instance of a real compressor. Cilibrasi has also made an implementation of the NCD available as the open source package CompLearn [11]. CompLearn implements the NCD using standard compression mechanisms, including *bzip2*, *gzip*, *lzma* and *ppmd*. In addition to this application in SE, the technique is also successfully used in genomics, plagiarism detection and detection of similarity of musical styles. (References for these can be found here [4].)

We have now seen how the Kolmogorov complexity measures the information content of objects and how, unlike the Shannon entropy, it has no requirement for difficult to obtain probability distributions. We have also seen how (mathematical) metrics for the comparison of objects have been derived from it. Further, we have seen how the uncomputable but exact relative information distance measure, the NID, can be approximated by the NCD which can be implemented by approximation involving real-world compression mechanisms. We now perform a Kolmogorov complexity-based software evolution experiment on real data.

3. PROOF-OF-CONCEPT EXAMPLE

In this section, we will characterise the evolution of the open source project *udev*, a key element of Linux system infrastructure. The aim of this short example is to demonstrate the utility of the approach by examining real-world data derived from a project of significant size.

The project *udev* currently comprises a suite of programs. Although the main functionality of the project is to dynamically generate filesystem nodes to represent hardware devices, particularly dynamically connected ones such as USB devices, it also includes functionality for the generation of unique hardware identifiers as well as configuration scripts which can be used to trigger or control actions corresponding to devices' connection or disconnection.

We chose this project because it is self-contained and has many well-documented releases: 0.1, 0.2, 003, 004 up to release 142. We assign these indices starting from 1. However, release 041 does not exist so index 40 corresponds to release 040 but index 41 corresponds to release 042, and so on. Although *udev* is written in C, we could equally well have chosen a program written in any other language.

3.1 Method

We need to derive input data from each release of the program as with any other method of examining software evolution. For this short example – which we present as evidence of plausibility – we proceed as follows.

1. From the distribution site (www.kernel.org), download all of the releases to be studied as compressed tar files.
2. Decompress and extract each archive.
3. In the decompressed archive's directory or any subdirectory thereof, concatenate all files found with the file extensions '.c' or '.h'. This creates a single file containing the unedited and unfiltered source which we retain as the data representative of the release.
4. For release indices x, y, calculate the distance matrix of all values of $NCD(x, y)$ by applying CompLearn's *ncd* using the *ppmd* compressor to the concatenated source files previously created.
5. Plot the distance matrix then compare the plot's significant features with the program's history or vice versa.

In this simple test, we employ the raw source code as the artefact for comparison. Clearly, since what we are examining is the pairwise, relative, number of bits of shared information, any artefact representative of the program will provide candidate data. We will further discuss this later.

3.2 Results

Figure 1 shows the distance matrix comparing all releases. The value of $NCD(x, y)$ is plotted in greyscale with black representing 0.0 and white representing the value 1.0. Overlaid on the plot are coloured contours corresponding to the values shown in the key. Self-comparisons along the diagonal are all zero (but occluded by the contour lines).

Remember that $1 - NCD(x, y)$ can be thought of as the number of bits of shared information between samples x and y proportional to the sample with more information. Therefore white areas, the top-left and bottom-right of the plot, correspond to comparisons between samples with little shared information. Moreover, since the releases are temporally sequential, moving from the diagonal in the directions left to right or bottom to top corresponds to comparing a given release (on the diagonal) with a later release.

Table 1: Reasons for major changes

Index	Release	Notable Features
6	6	SCCS files kept in source
16	16	Removal of SCCS files
43,44	44,45	No code changes
53	54	Update klibc with zlib
73	74	Remove own copy of klibc
79	80	Replace libsysfs
98,99	99,100	Almost no code changes
126	127	libudev info library

The blocks that can be seen along the main diagonal correspond to releases that are similar to each other. Wide reaching changes will be found at the corner points of each block along the diagonal. After a few initial releases (5), there is a set of around 10 releases that are very similar to each other. There is then another block of 57 releases followed by a block with the remaining releases. Within each of the two larger blocks, there are one large and one small sub-blocks: 17 to 53, 53 to 72; and 77 to 126, 126 to 141, respectively. Two dotted islands are present on the main diagonal at indices 43 and 98. The other feature of note is the dual 'spike' from releases 17 to 54 corresponding to comparison with release 5. One can say that release 5 is similar to release 17 onwards but not to releases 6 to 16. The releases at which major changes occurred are given in Table 1 together with the code alterations responsible for the large differences in shared information content.

This establishes the utility of the method. The plot told us where to look in the ChangeLog to see what the major changes were. Without a ChangeLog, we could have seen where major changes were introduced. Although we used source code here, would could equally well have employed obfuscated binaries for the test since we have not established what the most revealing artefacts for our purposes will be.

We note that there is some dependency on the quality of the compressors used in the approximation. A compressor better approximating the ideal compressor will give better results. Nevertheless, most commonly found compressors are suitable for use with the method (although see [6]).

4. CONCLUDING REMARKS

To provide a sound theoretical basis for software measurement, including the measurement of software evolution, we must look to information theory. Previous attempts to do so have almost all employed Shannon entropy as the basis for characterisation. Until recently, the alternative, Kolmogorov complexity, could not be explored because the tools were not available to do so. Since Kolmogorov complexity measures the information content of objects, we claim that it provides a more natural fit with the needs of software measurement. Further, it carries much of the meaning of Shannon entropy without the need for difficult to obtain probabilities and can thus be said to subsume much of the earlier work. The strong correlation between counting metrics and information content established by Allen *et al.* provides further justification for this choice.

We have experimentally demonstrated plausibility here by investigating raw source code as the artefact for tracing software evolution. Investigating behaviour in terms of dynamically produced traces [4] or structure in terms of graph representations of software [3] have also been previously outlined. (We intuit connections with sequence-based specification techniques [29].) We have sketched that the idea of employing Kolmogorov complexity as the basis for software measurement is worthy of further investigation since the limits and desirable context are unknown. The remaining task amounts to the discovery of which artefacts – or kinds of artefacts – are most representative of software and software development operations and why.

5. ACKNOWLEDGMENTS

The author would like to thank: Jim Buckley for helpful advice and reviews; Mark Lawford for encouragement and practical assistance; and the anonymous reviewers for their constructive comments. Adam Balaban and Dennis K. Peters have previously provided support and insight.

6. REFERENCES

[1] C. Alexander. *Notes on the Synthesis of Form.* Harvard U. Press, 1964.

[2] E. B. Allen, S. Gottipati, and R. Govindarajan. Measuring size, complexity, and coupling of hypergraph abstractions of software: An information-theory approach. *Software Quality Journal*, 15(2):179–212, 2007.

[3] T. Arbuckle. Visually summarising software change. In *Proc. 12th International Conference Information Visualisation*, pages 559–568, 2008.

[4] T. Arbuckle, A. Balaban, D. K. Peters, and M. Lawford. Software documents: Comparison and measurement. In *Proc. 18th Int. Conf. on Software Eng. & Knowledge Eng.*, pages 740–745, July 2007.

[5] C. H. Bennett, P. Gács, M. Li, P. Vitányi, and W. H. Zurek. Information distance. *IEEE Trans. Information Theory*, 44(4):1407–1423, 1998.

[6] M. Cebrián, M. Alfonseca, and A. Ortega. Common pitfalls using the normalized compression distance: What to watch out for in a compressor. *Comms. Info. Sys.*, 5(4):367–384, 2005.

[7] G. J. Chaitin. On the length of programs for computing finite binary sequences: statistical considerations. *J. ACM*, 16(1):145–159, 1969.

[8] R. N. Chanon. *On a measure of program structure.* PhD thesis, Carnegie-Mellon University, 1974.

[9] R. N. Chanon. On a measure of program structure. In *Programming Symposium, Proceedings Colloque sur la Programmation*, pages 9–16. Springer-Verlag, 1974.

[10] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Trans. Softw. Eng.*, 20(6):476–493, 1994.

[11] R. Cilibrasi. The CompLearn Toolkit. [Online] http://complearn.sourceforge.net/, 2003.

[12] R. Cilibrasi and P. Vitányi. Clustering by compression. *IEEE Trans. Information Theory*, 51(4):1523–1545, April 2005.

[13] D. Clark, S. Hunt, and P. Malacaria. Quantitative information flow, relations and polymorphic types. *JLC: Journal of Logic and Computation*, 15, 2005.

[14] N. Fenton. When a software measure is not a measure. *Softw. Eng. J.*, 7(5):357–362, 1992.

[15] M. H. Halstead. *Elements of Software Science.* Elsevier Science Inc., 1977.

[16] M. Harman. The current state and future of search based software engineering. In *FOSE '07: 2007 Future of Software Engineering*, pages 342–357, 2007.

[17] M. Harman. Search based software engineering for program comprehension. In *ICPC '07: 15th Int. Conf. on Program Comprehension*, pages 3–13, 2007.

[18] L. Hellerman. A measure of computational work. *IEEE Trans. Computers*, C-21(5):439–446, May 1972.

[19] M. Jackson. *The Name and Nature of Software Engineering*, pages 1–38. LNCS. 2008.

[20] D. Kafura. A survey of software metrics. In *ACM '85: Proc. 1985 ACM annual conference on The range of computing : mid-80's perspective*, pages 502–506. ACM Press, 1985.

[21] H. Kagdi, M. L. Collard, and J. I. Maletic. A survey and taxonomy of approaches for mining software repositories in the context of software evolution. *J. Softw. Maint. Evol.*, 19(2):77–131, 2007.

[22] T. M. Khoshgoftaar and E. B. Allen. Applications of information theory to software engineering measurement. *Software Quality Journal*, 3(2):79–103, June 1994.

[23] A. N. Kolmogorov. Three approaches to the quantitative definition of information. *Probl. Inform. Trans.*, 1(1):1–7, 1965.

[24] G. Kroah-Hartman and K. Sievers. udev. [Online] http://www.kernel.org/, 2003.

[25] M. Li, X. Chen, X. Li, B. Ma, and P. Vitányi. The similarity metric. *IEEE Trans. Information Theory*, 50(12):3250–3264, 2004.

[26] R. Lutz. Evolving good hierarchical decompositions of complex systems. *Journal of systems architecture*, 47(7):613–634, 2001.

[27] T. J. McCabe. A complexity measure. In *ICSE '76: Proceedings of the 2nd international conference on Software engineering*, page 407, 1976.

[28] S. McCamant and M. D. Ernst. Quantitative information flow as network flow capacity. In R. Gupta and S. P. Amarasinghe, editors, *PLDI*, pages 193–205. ACM, 2008.

[29] S. J. Prowell and J. H. Poore. Foundations of sequence-based software specification. *IEEE Trans. Softw. Eng.*, 29(5):417–429, 2003.

[30] C. E. Shannon. A mathematical theory of communication. *The Bell System Technical Journal*, 27:379–423 and 623–656, 1948.

[31] H. A. Simon and A. Ando. Aggregation of variables in dynamic systems. *Econometrica*, 29:111–138, 1961.

[32] R. J. Solomonoff. A formal theory of inductive inference. part I and part II. *Information and Control*, 7(1 and 2):1–22 and 224–254, 1964.

[33] M. H. van Emden. Hierarchical decomposition of complexity. *Machine Intelligence*, 5:361–380, 1969.

[34] M. H. van Emden. *An Analysis of Complexity.* PhD thesis, Mathematisches Zentrum, Amsterdam, 1971.

[35] H. Zuse. *Software Complexity: Measures and Methods.* Walter de Gruyter & Co., Hawthorne, NJ, USA, 1990.