Software Documents: Comparison and Measurement

Tom Arbuckle*, Adam Balaban†, Dennis K. Peters‡ and Mark Lawford§
*Department of Computer Science and Information Systems, College of Informatics and Electronics,
CSIS Building, University of Limerick, Plassey Park, Limerick, Ireland. Email: tom.arbuckle@ieee.org
†Institute of Informatics, Warsaw University, Banacha 2, 02-097 Warsaw, Poland. Email: ab@mimuw.edu.pl
‡Electrical and Computer Engineering, Faculty of Engineering and Applied Science,
Memorial University of Newfoundland, St. John's NL, Canada A1B 3X5. Email: dpeters@engr.mun.ca
§Department of Computing and Software, Faculty of Engineering, McMaster University,
Hamilton, Ontario, Canada, L8S 4K1. Email: lawford@mcmaster.ca

***Abstract*— For some time now, researchers have been seeking to place software measurement on a more firmly grounded footing by establishing a theoretical basis for software comparison. Although there has been some work on trying to employ information theoretic concepts for the quantification of code documents, particularly on employing entropy and entropy-like measurements, we propose that employing the Similarity Metric of Li, Vitányi, and coworkers for the comparison of software documents will lead to the establishment of a theoretically justifiable means of comparing and evaluating software artifacts. In this paper, we review previous work on software measurement with a particular emphasis on information theoretic aspects, we examine the body of work on Kolmogorov complexity (upon which the Similarity Metric is based), and we report on some experiments that lend credence to our proposals. Finally, we discuss the potential advantages derived from the application of this theory to areas in the field of software engineering.**

I. Introduction

In the transition from the idea or concept for an element of software to be implemented to the code that implements it and beyond, it is recognised that there are several (possibly overlapping) stages (perhaps labeled by analysis, design, coding, testing and support). In the production of quality software, we assume that these stages — however they are assigned — are documented. For the purposes of evaluation and assessment, we therefore need to be able to compare documents that describe or specify [1] — or indeed implement — software.

In software engineering, the comparison or measurement of software documents (including source code and executable objects) has traditionally been assigned to the subfield of software measurement – often called 'metrics'. The task of characterising something that is effectively infinitely malleable has been frought with difficulty. It has been difficult to agree what to measure and the suitability of what is measured, and to validate the measures against the implementations. Thus the field has largely become a means of providing indicators, symptoms to be evaluated by experienced practitioners rather than a means of objectively measuring attributes of software and their significance. We believe that the Similarity Metric (of Li, Vitányi and others) can help in this regard. Kolmogorov complexity, on which the measure is based, in a very precisely defined sense describes the inherent complexity of objects. Since the documents produced to describe, specify or implement code are also 'objects' which embody their intended meaning, we regard them as being inherently suitable comparands for this method.

To be more specific, we wish to examine ways in which the Similarity Metric can be employed in software engineering. This will include the comparison of specifications or descriptions of software before the software is implemented.

This paper is structured as follows. In the next sections, we introduce software measurement, information theory and Kolmogorov complexity. There follows an overview of the work, largely by Li and Vitányi, on the definition of a universal comparator, the Similarity Metric [2], whose foundation is the complexity theory of Kolmogorov. In the next section, we will attempt to relate complexity and measurement from the viewpoint of the field of software engineering, in particular software measurement. After a section in which we apply Cilibrasi's implementation [3] of an approximation of the Similarity Metric [4] to a set of software engineering examples, we discuss what we believe to be the significance of these techniques to the software engineering field and outline additional applications beyond those detailed herein. We close with our conclusions and acknowledgements.

II. Software Measurement

A. Introduction

The field of software measurement has a long history [5], [6], [7] and a broad literature. We can measure software to examine its performance; its structure or design; its correctness; its quality; its evolution (in terms of maintenance or extension); the processes executed by its creators. Broadly speaking, the field is important because only by performing measurements (however, defined) on software and comparing those measurements, can we make objective statements about these topics. For a good overview of the field of software measurement, see Fenton and Pfleeger [8].

B. Software Engineering: Metrics and Measures

Largely in agreement with Zuse [9], we define the words 'measure', 'measurement' and 'metric' as follows.

- A *measure* is a mapping from empirical objects (objects to be measured) to formal objects (measurement values). We will define a *measurement* to be the result of applying a 'measure'.

- A *metric* is a means of determining the distance between two entities. A metric function (of two arguments) must also (1) return null for identical entities; (2) be symmetric; and (3) obey the triangle inequality.

In the literature, different definitions for these words are common. In particular, in software engineering, other conventions are often followed. Lorenz and Kidd [10], for example, make the following two definitions.

- *Metric* A standard for measurement. Used to judge the attributes of something being measured such as quality or complexity, in an objective manner.
- *Measurement* The determination of the value of a metric for a particular object.

Our definitions are more in agreement with the mathematics literature but potential for confusion should be borne in mind.

We will not discuss further the extensive literature concerning definitions, axiomatic approaches or properties that software 'metrics' might have [11], [12], [13], [14], [15].

III. Information Theory, Entropy

In his 1948 paper [16], Shannon was concerned with the transmission of information from a source to a sink over a channel. He considered encoding of information, discrete and continuous encodings, and transmission in the presence or absence of noise. As a measure of the 'quantity' of information to be passed through a channel, he introduced entropy, an analogue of the thermodynamic entropy of Boltzmann. In the discrete case, (Shannon) entropy takes its well-known form

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

where $H(X)$ is the entropy for a source emitting codes x from a set X, and the p_x are the probabilities the codes x occur.

It can be claimed that Shannon's paper marked the founding of the field of information theory. Certainly, an indication of the quantity of information being produced by a source is a useful concept that has since been applied in many fields [17].

IV. Kolmogorov Complexity

To quote from the book [18] by Li and Vitányi, "Shannon's entropy measures the uncertainty in a statistical ensemble of messages, while Kolmogorov complexity measures the algorithmic information in an individual message." Also sometimes known as 'algorithmic entropy', Kolmogorov complexity was introduced independently by Solomonoff [19], Kolmogorov [20], and Chaitin [21].

More formally, the Kolmogorov complexity, $K(x)$ of a binary string x is the length of the shortest (prefix-free) binary program to compute x on a universal computer such as a universal Turing machine. $K(x)$ represents the number of bits necessary to (computationally) describe the string x.

Although this definition sounds somewhat abstract, there are indeed strong connections with the Shannon entropy [22], [23]. It can be shown, for example, that the expected value of the Kolmogorov complexity for a random string for any probability distribution function will have a value to within an additive constant (dependent only on the executing Turing machine) of the Shannon entropy.

As might be expected, there are also connections between Kolmogorov complexity and thermodynamics. Bennett *et al.* [24] describe how it can be related to the thermodynamic cost (minimal entropy increase in the environment) of data transformations. This follows from early work on thermodynamics in computing by Landauer [25].

V. Measuring Similarity

A. Information Distance

The idea of being able to calculate a value for the complexity of objects leads naturally to the idea of being able to compare how similar objects are to each other. In their 1998 paper, Bennett *et al.* [26] investigated the idea of 'an "absolute information distance metric" between individual objects' that they have subsequently shown [27] to obey the (mathematical) metric properties up to an additive constant. The information distance was to form the basis of the subsequent normalised similarity metric. Here, normalisation means that $0 \leq d(x, y) \leq 1$ with 0 indicating identical objects.

B. The Similarity Metric

A first attempt at a normalised similarity metric was presented by Li *et al.* [28]. However, in their 2004 paper [27], Li *et al.* presented an improved version, the normalised information distance (NID) given by

$$NID(x, y) = \frac{max\{K(x|y), K(y|x)\}}{max\{K(x), K(y)\}} \tag{2}$$

Here $K(x|y)$, for example, is the conditional Kolmogorov complexity of x given y. This is the length of the shortest program for a universal Turing machine to output x when given an input y. They showed that this new distance satisfies the metric properties up to an additive term of $O(1/K)$, where K is the maximum of Kolmogorov Complexities involved. Then $1 - NID(x, y)$ has the natural interpretation of the number of bits of shared information per bit of information of the string with more information.

C. Practical Implementation — CompLearn

Since the idea of the NID concerns compression of data, the normalised information distance can be approximated by using real compressors, including most commonly known compressors, which obey certain properties (idempotency, monotonicity, symmetry and distributivity) [4]. The resulting measure is called the normalised compression distance (NCD).

$$NCD(x, y) = \frac{C(xy) - min\{C(x), C(y)\}}{max\{C(x), C(y)\}} \tag{3}$$

where $C(x)$, 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 and xy denotes the concatenation of x and y.

Although the theory for the NID is exact, additional theoretical work was carried out by Cilibrasi and Vitányi [4]

to show that the NCD was a good approximation whose difference from the NID was dependent only on the quality of the approximation of a 'perfect' compressor by a real one.

An implementation of the NCD has been made publicly available as the CompLearn toolkit [3]. We have employed version 0.9.7 of the toolkit in the experiments that follow. The toolkit compressors we employ are the Lempel-Ziv zlib algorithm, the bzip2 block-sorting compressor and the blocksort algorithm provided in CompLearn by Cilibrasi.

VI. Software, Complexity, Information Theory

Campbell [29] was already considering whether entropy could be used as a metric (in the mathematical sense) as early as 1965. A 1972 paper by Hellerman [30] employed entropy to measure information in a computer's memory as a measure of computational work. However, possibly the first use of entropy in the discussion of design was van Emden's 1969 paper [31] and his later thesis [32]. Van Emden's concern was the use of a form of entropy (called "surplus entropy", or "entropy loading") to decide how to decompose an object into subcomponents. This work was taken up by Chanon who showed [33], [34] how van Emden's work could be employed for the structuring of software. See also [35], [36], [37].

Since this early work, numerous authors have sought to apply information theory concepts to software engineering. A full review would be a paper in its own right. To list some examples, judging the information-theoretic complexity of software specifications was studied by Coulter *et al.* [38]. Bansiya *et al.* [39] modeled the complexity of object-oriented systems using entropy. (See Tegarden *et al.* for a 'metrics' approach [40].) 'Coupling', the degree of interconnection or dependency between software components, and 'cohesion', the degree of internal interconnection, were modeled in information theoretic terms by Allen *et al.* [41], [42]. There are also studies of apportioning complexity [43], [44] and of the correlation between forms of code complexity and the likelihood of errors [45], [46].

Research on using information theory based methods for quantifying software continues, a recent paper by Sarkar *et al.* [47] being one good example.

VII. Examples

A. Experiments in other fields

There should be no doubt that Cilibrasi's implementation of Li and Vitányi's Similarity Metric has already proven its utility and correctness. Amongst others, it has been applied to program plagiarism detection [48], genomics [4], cross-language textual similarity [49], and the classification of musical styles [50]. Although a subsequent paper by Vitányi [51] states that the program has also been applied to the classification of programs written in the languages Ruby and C, we have not found details of these experiments. In addition, we are unaware of any descriptions of applications of these techniques in the field of software engineering.

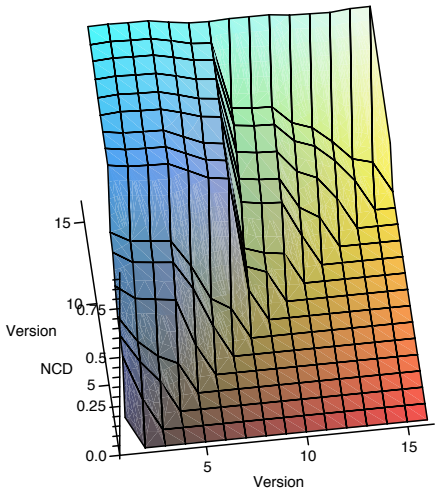

Fig. 1. *ncd* operating on sources

B. Sample Experiment — The 'slocate' package

The open-source software package *slocate* [52] is designed to catalogue and index (as the superuser) all files present in a specified area of a filesystem but to answer queries such that the visibility of files is filtered by the authorisation privileges of the current software user. *slocate* is an ideal candidate for one of our experiments: its code is short but performs fairly complex operations; it has a long history with many publicly available releases; and it has also been redesigned on at least one occasion. Reference indices (for use in diagrams) and source lengths in kilobytes for the releases of *slocate* that we studied are as shown in table I.

TABLE I

slocate RELEASES, INDICES, SIZES (KILOBYTES)

Rel.	1.0	1.1	1.2	1.3	1.4	1.5	1.6	2.0	2.1	2.2	2.3	2.4	2.5	2.6	2.7	3.1
Ind.	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Size	26.6	26.7	24.0	23.9	29.7	31.3	32.2	70.3	72.3	72.7	70.7	73.6	82.6	91.3	92.0	67.4

1) Source Code: For each version, we applied the *ncd* program from CompLearn to the concatenation of all ".h" and ".c" files whose lengths were shown in table I. A plot of (half of) the NCD similarity matrix is shown in figure 1.

2) Indexing Trace: Using the *strace* program which provides a listing of calls to system functions from user programs, we created a trace (sequence of program calls) of the operation of the versions of the *slocate* program creating a database on the same data. Figure 2 shows a plot of (half of) the NCD similarity values.

3) Search Trace: The *strace* program was again used to produce a trace of system calls, this time for answering the same search query on the database produced in the previous experiment. Figure 3 shows the plot of (half of) the results of running *ncd* on these traces.

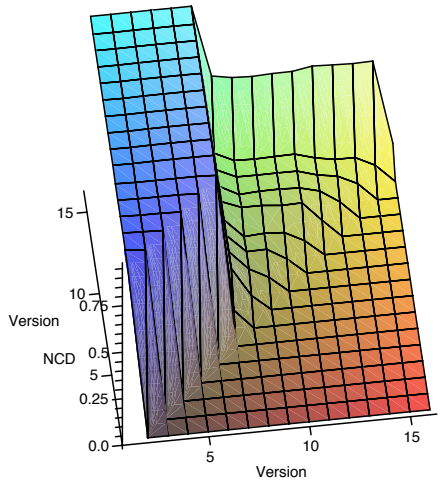

Fig. 2. *ncd* on indexing trace

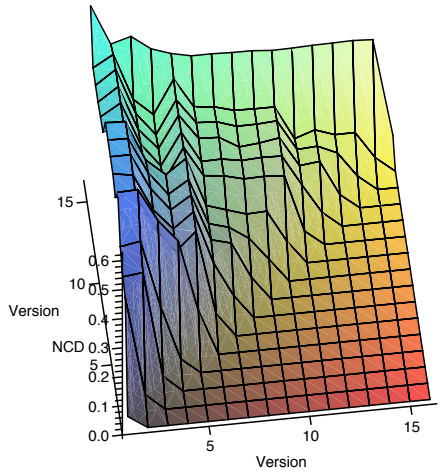

Fig. 3. *ncd* on search trace

C. Analysis

Each of the figures shows a pairwise similarity between different versions of *slocate*. The *ncd* similarity matrix is (almost) symmetric and so values below (not on) the main diagonal of the matrix have been zeroed to permit easier viewing. Remember that NCD values close to zero denote similarity and those close to one denote strong dissimilarity.

Let us examine figure 1. From left to right, the columns represent the NCD values for increasing versions. For indices 1 to 7, we see a steep increase in the degree of dissimilarity with increasing versions so that, above index 8, the degree of dissimilarity forms a plateau in the top-left corner. From index 8 onwards, we see (up the columns) increasing degrees of dissimilarity leading to a 'back wall' for the final index.

This diagram tells us that the author was justified in his versioning system since the discontinuities correspond to major releases (2.0, 3.1). We can also see the gradual differences between minor versions. We can claim that this is evidence of good incremental design. We can further claim that this is evidence of good structure since the changes introduced between minor versions were permitted by the design without engendering major modifications.

Figure 2 shows the results of comparing the (system) operations for building an index with each version of *slocate* being run on the same input. Differences between operations for minor releases of the 1.x code are very apparent. Differences in operations for the 2.x series are more gradual and there appears to be some similarity to the operations of the 3.x series (forming the 'back wall' of the figure). We see that NCD is detecting the right differences for our purposes.

Now examine figure 3 showing the result of a search on the respective databases. Looking along the columns, indices 1 and 2 show that the first two versions behave differently from the others. We can see another 'ridge' at index 5. The two previous versions were 'optimising' the code whereas this version adds functionality (globbing, greater compatibility with GNU *locate*). We can also see a plateau between (horizontal) indices 6 and 10 for indices 11 to 15 indicating that the final releases of the 2.x series behave differently from both the 1.x series and earlier versions of the 2.x series. Again, we note the presence of the 'back wall' in the diagram. The operations performed by version 3.1 are clearly different from those of the 2.x series.

Finally, compare the diagrams with each other. We see clear similarities between figures 1 and 2 and clear differences between these two and figure 3. The building of the database mirrors the versioning with little work being done on it between minor releases. The bulk of the change we see is in the search of the database even within versions.

Suppose a developer has data about their current project similar to that presented here. The developer is aware of the changes being made but may not be so aware of their repercussions. By comparing current behaviour with that of previous versions, the developer can see where a possibly small change in the code results in a much larger change in behaviour (that might not be immediately apparent).

These plots are clearly of great utility to both managers and coders alike. For minimal investment of time and energy, it is easy to see important aspects of change throughout the version history of the project. The successful application of NCD to *strace* logs we consider very promising but there are many other applications for this technique.

The recent trace function method (TFM) [53], a successor of the earlier trace assertion method [54], permits software specification in terms of traces as do other formalisms such as enumerative specification [55], [56], path expressions [57] and the work of Broy [58]. This suggests the following.

Firstly, using simulation techniques applied to those specifications we can generate traces for the specified software. Whereas our earlier experiments required working code and

the *strace* program, we can obtain the data for our comparisons at the design stage thereby permitting the measurement of the development progress and of the complexity of the software before the implementation work has begun. We believe that even the design should be done in incremental steps and these tools will be helpful in measuring this. Given a sufficient body of data, design complexity can be validated and properties of future designs quantified before coding begins.

Secondly, Peters *et al.* [59] are seeking to encode program specifications written using TFM in OMDoc [60] XML files. These encodings of software specifications exist before the implementations the specifications describe. Descriptions of existing software can also be created using this encoding. In future experiments, we will be interested to examine OMDoc descriptions or specifications of software.

VIII. Discussion, Future Work

The work of Li, Vitányi and Cilibrasi provides a fundamentally new and theoretically justified approach that we can apply to our problems. Its application in the field of software engineering presents many opportunities to help practitioners do a better job of creating quality software. There are numerous additional applications of the techniques that we have shown.

- We can automate the monitoring of the evolution of code as it moves through different phases of implementation, testing and maintenance. This will allow us to track the introduction of additional complexity or to see simplifications made without loss in functionality.
- Once an interface for an implementation is specified, application of these methods will permit the comparison of different implementations. We can compare internal designs for different modules and identify similar modules as potential refactoring candidates.
- Given some estimate of the complexity of a design, we will be better able to make management decisions about the quantities of resources required to complete a given design or programming task.
- Given a specification, we can implement that specification in many possible programming languages. If we can find a common representation of their execution output, we can compare different languages to see if they aid or hinder the creation of implementations.
- By measuring the modules that comprise the implementation, we will be able to make estimates of the relative complexities of the code for those units. Since error density has been shown to correlate with code complexity, we will have better ideas of where to concentrate our testing efforts and of where errors are likely to occur.
- Gathering requirements and creating usage scenarios allows us to decide what needs to be created and to evaluate whether a given task is viable. Given a suitable encoding, the comparison of different sets of requirements for the same project would permit some evaluation of relative complexities before going on to design.
- The work of different coders or designers working to a common specification can be compared using this method. Planning for future change, a more experienced designer might produce a more complex design. However, even the ability to reveal this is of great utility.
- In addition we intuit connections with considerations of coverage [61] that we would like to investigate.

We are aware that these suggestions do have some drawbacks. The interpretation of these comparisons requires some thought. An attempt to compare a 'large' set of documents might fail for practical reasons. More importantly, however, the question of the coding of the documents needs to be considered. Vitányi has stated [51] that while the technique is robust with respect to a change in underlying compressor types (although see [62]), there are still situations in which a naïve application of the technique may fail. He states that further research is necessary to examine the case where the input strings are overly sensitive to the encoding used. Nevertheless, we foresee that applying the Similarity Metric in software measurement will provide many areas for future exploration and are highly encouraged by our results so far.

IX. Conclusion

In this paper, we have shown that the application of the similarity metric of Li and Vitányi (based on Kolmogorov complexity theory) in the field of software engineering will provide a theoretically sound basis on which to found software measurements. We claim that having a theoretically justified means of comparison of software documents, at many stages in the software lifecycle, will have a beneficial effect on the reproducibility and justifiability of measurement claims. We have shown that using this method can provide useful information for software engineers during design, testing and maintenance. We have described several ways in which this theory can be applied and, in future publications, we will elaborate on our suggestions and their practical utilisation.

Acknowledgments

The authors thank numerous reviewers for their comments on earlier drafts. This research is supported by Science Foundation Ireland (grants 01/P1.2/C009 and 03/CE3/1405).

References

[1] D. L. Parnas, "Precise description and specification of software," in *MDS '95: Proc. 2nd international conference on Mathematics of dependable systems II*. Oxford University Press, Inc., 1997, pp. 1–14.

[2] M. Li, X. Chen, X. Li, B. Ma, and P. Vitányi, "The similarity metric," in *SODA '03: Proceedings of the fourteenth annual ACM-SIAM symposium on Discrete algorithms*. SIAM, 2003, pp. 863–872.

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

[4] R. Cilibrasi and P. Vitányi, "Clustering by compression," *IEEE Trans. Information Theory*, vol. 51, no. 4, pp. 1523–1545, April 2005.

[5] R. J. Rubey and R. D. Hartwick, "Quantitative measurement of program quality," in *Proceedings of the 1968 23rd ACM national conference*. New York, NY, USA: ACM Press, 1968, pp. 671–677.

[6] M. H. Halstead, *Elements of Software Science (Operating and programming systems series)*. New York, USA: Elsevier Science Inc., 1977.

[7] B. H. Yin and J. W. Winchester, "The establishment and use of measures to evaluate the quality of software designs," in *Proceedings of the software quality assurance workshop on Functional and performance issues*, 1978, pp. 45–52.

[8] N. E. Fenton and S. L. Pfleeger, *Software Metrics: A Rigorous and Practical Approach*. Boston, MA, USA: PWS Publishing Co., 1998.

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

[10] M. Lorenz and J. Kidd, *Object-oriented software metrics: a practical guide*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1994.

[11] N. Fenton, "Software measurement: A necessary scientific basis," *IEEE Trans. Softw. Eng.*, vol. 20, no. 3, pp. 199–206, 1994.

[12] B. Henderson-Sellers, "The mathematical validity of software metrics," *SIGSOFT Softw. Eng. Notes*, vol. 21, no. 5, pp. 89–94, 1996.

[13] E. J. Weyuker, "Evaluating software complexity measures," *IEEE Trans. Softw. Eng.*, vol. 14, no. 9, pp. 1357–1365, 1988.

[14] L. C. Briand, S. Morasca, and V. R. Basili, "Property-based software engineering measurement," *IEEE Trans. Softw. Eng.*, vol. 22, no. 1, pp. 68–86, 1996.

[15] N. Fenton, "When a software measure is not a measure," *Softw. Eng. J.*, vol. 7, no. 5, pp. 357–362, 1992.

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

[17] T. M. Cover and J. A. Thomas, *Elements of Information Theory*. Wiley-Interscience, 2006.

[18] M. Li and P. Vitányi, *An introduction to Kolmogorov complexity and its applications (2nd ed.)*. Springer-Verlag, 1997.

[19] R. J. Solomonoff, "A formal theory of inductive inference. part I and part II," *Information and Control*, vol. 7, no. 1 and 2, pp. 1–22 and 224–254, 1964.

[20] A. N. Kolmogorov, "Three approaches to the quantitative definition of information," *Probl. Inform. Trans.*, vol. 1, no. 1, pp. 1–7, 1965.

[21] G. J. Chaitin, "On the length of programs for computing finite binary sequences: statistical considerations," *J. ACM*, vol. 16, no. 1, pp. 145–159, 1969.

[22] P. D. Grünwald and P. M. B. Vitányi, "Kolmogorov complexity and information theory with an interpretation in terms of questions and answers," *J. of Logic, Lang. and Inf.*, vol. 12, no. 4, pp. 497–529, 2003.

[23] ——, "Shannon information and Kolmogorov complexity," September 2004, submitted to IEEE Trans. Information Theory.

[24] C. H. Bennett, P. Gács, M. Li, P. M. B. Vitányi, and W. H. Zurek, "Thermodynamics of computation and information distance," in *STOC '93: Proceedings of the twenty-fifth annual ACM symposium on Theory of computing*. New York, NY, USA: ACM Press, 1993, pp. 21–30.

[25] R. Landauer, "Irreversibility and heat generation in the computing process," *IBM J. Res. Develop.*, vol. 5, no. 3, pp. 183–191, July 1961.

[26] C. H. Bennett, P. Gács, M. Li, P. Vitányi, and W. H. Zurek, "Information distance," *IEEE Transactions on Information Theory*, vol. 44, no. 4, pp. 1407–1423, July 1998.

[27] M. Li, X. Chen, X. Li, B. Ma, and P. Vitányi, "The similarity metric," *IEEE Transactions on Information Theory*, vol. 50, no. 12, pp. 3250–3264, December 2004.

[28] M. Li, J. H. Badger, X. Chen, S. Kwong, P. Kearney, and H. Zhang, "An information-based sequence distance and its application to whole mitochondrial genome phylogeny." *Bioinformatics*, vol. 17, no. 2, pp. 149–154, 2001.

[29] L. L. Campbell, "Entropy as a measure," *IEEE Trans. Information Theory*, vol. 11, no. 1, pp. 112–11, 1965.

[30] L. Hellerman, "A measure of computational work," *IEEE Trans. Computers*, vol. C-21, no. 5, pp. 439–446, May 1972.

[31] M. H. van Emden, "Hierarchical decomposition of complexity," *Machine Intelligence*, vol. 5, pp. 361–380, 1969.

[32] ——, "An analysis of complexity," Ph.D. dissertation, Mathematisches Zentrum, Amsterdam, 1971.

[33] R. N. Chanon, "On a measure of program structure," in *Programming Symposium, Proceedings Colloque sur la Programmation*. London, UK: Springer-Verlag, 1974, pp. 9–16.

[34] ——, "On a measure of program structure." Ph.D. dissertation, Carnegie-Mellon University, 1974.

[35] S. Henry and D. Kafura, "Software structure metrics based on information flow," *IEEE Transactions on Software Engineering*, vol. 7, no. 5, pp. 510–518, September 1981.

[36] N. Chapin, "An entropy metric for software maintainability," in *Proceedings of the Twenty-Second Anuual Hawaii International Conference on System Sciences*, vol. II, Jan 1989, pp. 522–533.

[37] W. R. Torres and M. H. Samadzadeh, "Software reuse and information theory based metrics," in *Proc. 1991 Symposium on Applied Computing*, April 1991, pp. 437–446.

[38] N. S. Coulter, R. B. Cooper, and M. K. Solomon, "Information-theoretic complexity of program specifications," *Comput. J.*, vol. 30, no. 3, pp. 223–227, 1987.

[39] J. Bansiya, C. Davis, and L. Etzkorn, "An entropy-based complexity measure for object-oriented designs," *Theor. Pract. Object Syst.*, vol. 5, no. 2, pp. 111–118, 1999.

[40] D. P. Tegarden, S. D. Sheetz, and D. E. Monarchi, "A software complexity model of object-oriented systems," *Decis. Support Syst.*, vol. 13, no. 3-4, pp. 241–262, 1995.

[41] E. B. Allen and T. M. Khoshgoftaar, "Measuring coupling and cohesion: An information-theory approach," in *METRICS '99: Proc. 6th Int. Symp. on Software Metrics*. IEEE Computer Society, 1999, p. 119.

[42] E. B. Allen, T. M. Khoshgoftaar, and Y. Chen, "Measuring coupling and cohesion of software modules: An information-theory approach," in *METRICS '01: Proceedings of the 7th International Symposium on Software Metrics*. IEEE Computer Society, 2001, p. 124.

[43] J. C. Munson and T. M. Khoshgoftaar, "The dimensionality of program complexity," in *ICSE '89: Proceedings of the 11th international conference on Software engineering*. ACM Press, 1989, pp. 245–253.

[44] T. M. Khoshgoftaar and D. L. Lanning, "Are the principal components of software complexity data stable across software products?" in *Proc. 2nd Int. Software Metrics Symposium*, 1994, pp. 61–72.

[45] D. L. Lanning and T. M. Khoshgoftaar, "Modeling the relationship between source code complexity and maintenance difficulty," *Computer*, vol. 27, no. 9, pp. 35–40, 1994.

[46] N. E. Gold, A. M. Mohan, and P. J. Layzell, "Spatial complexity metrics: An investigation of utility," *IEEE Trans. Softw. Eng.*, vol. 31, no. 3, pp. 203–212, 2005.

[47] S. Sarkar, G. M. Rama, and A. C. Kak, "API-based and information-theoretic metrics for measuring the quality of software modularisation," *IEEE Trans. Softw. Eng.*, vol. 33, no. 1, pp. 14–32, 2007.

[48] X. Chen, B. Francia, M. Li, B. McKinnon, and A. Seker, "Shared information and program plagiarism detection," *IEEE Trans. Information Theory*, vol. 50, no. 7, pp. 1545–1551, July 2004.

[49] K. Koroutchev and M. Cebrián, "Detecting translations of the same text and data with common source," *Journal of Statistical Mechanics: Theory and Experiment*, vol. 10, October 2006.

[50] R. Cilibrasi, P. Vitányi, and R. de Wolf, "Algorithmic clustering of music based on string compression," *Computer Music Journal*, vol. 28, no. 4, pp. 49–67, 2004.

[51] P. Vitányi, "Universal similarity," in *Proc. IEEE ISOC ITW2005 on Coding and Complexity*, M. Dinneen, Ed., 2005, pp. 238–243.

[52] K. Lindsay, "slocate," 1998, Canonical URL is http://slocate.trakker.ca/. [Online]. Available: http://freshmeat.net/projects/slocate/

[53] D. L. Parnas and M. Dragomiroiu, "Module Interface Documentation – Using the Trace Function Method (TFM)," 2006, submitted to *IEEE Trans. Softw. Eng.*

[54] W. Bartussek and D. L. Parnas, "Using traces to write abstract specifications for software modules," in *Proc. 2nd Conf. European Cooperation in Informatics*, ser. LNCS 65. Springer-Verlag, 1978, pp. 211–236.

[55] S. J. Prowell, "Developing black box specifications through sequence enumeration," in *SESD '99: Proceedings of the Science and Engineering for Software Development: A Recognition of Harlan D. Mills' Legacy*. Washington, DC, USA: IEEE Computer Society, 1999, p. 14.

[56] S. J. Prowell and J. H. Poore, "Foundations of sequence-based software specification," *IEEE Trans. Softw. Eng.*, vol. 29, no. 5, pp. 417–429, 2003.

[57] S. Andler, "Predicate path expressions," in *POPL '79: Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. New York, NY, USA: ACM Press, 1979, pp. 226–236.

[58] M. Broy and I. Krüger, "Interaction interfaces - towards a scientific foundation of a methodological usage of message sequence charts," in *ICFEM '98: Proc. 2nd IEEE Int. Conf. on Formal Engineering Methods*. IEEE Computer Society, 1998, pp. 2–13.

[59] D. K. Peters, M. Lawford, and B. T. y Widemann, "Software specification using tabular expressions and OMDoc," 2007, Work in progress.

[60] M. Kohlhase, *OMDoc — An Open Markup Format for Mathematical Documents [V. 1.2]*, ser. LNAI 4180. Berlin, Germany: Springer, 2006.

[61] G. H. Walton and J. H. Poore, "Measuring complexity and coverage of software specifications," *Information and Software Technology*, vol. 42, pp. 859–872, 2000.

[62] 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.*, vol. 5, no. 4, pp. 367–384, 2005.