Centrum voor Wiskunde en Informatica

REPORT*RAPPORT*

SEN

Software Engineering

Software ENgineering

Quantification of structural information: On a question raised by Brooks

J. Heering

REPORT SEN-E0311 DECEMBER 8, 2003

CWI is the National Research Institute for Mathematics and Computer Science. It is sponsored by the Netherlands Organization for Scientific Research (NWO).
CWI is a founding member of ERCIM, the European Research Consortium for Informatics and Mathematics.

CWI's research has a theme-oriented structure and is grouped into four clusters. Listed below are the names of the clusters and in parentheses their acronyms.

Probability, Networks and Algorithms (PNA)

Software Engineering (SEN)

Modelling, Analysis and Simulation (MAS)

Information Systems (INS)

P.O. Box 94079, 1090 GB Amsterdam (NL)
Kruislaan 413, 1098 SJ Amsterdam (NL)
Telephone +31 20 592 9333
Telefax +31 20 592 4199

ISSN 1386-369X

Quantification of structural information: On a question raised by Brooks

ABSTRACT
We introduce the notion of "generative software complexity" to illustrate some of the problems one may run into when trying to tackle a special case of a question recently raised by Brooks.

1998 ACM Computing Classification System: D.2.8
Keywords and Phrases: structural complexity of software; quantification of structural information
Note: Published in ACM SIGSOFT Software Engineering Notes 28(3) (May 2003)

Quantification of Structural Information: On a Question Raised by Brooks

Jan Heering
CWI
Kruislaan 413
1098 SJ Amsterdam
The Netherlands
e-mail: Jan.Heering@cwi.nl
www: www.cwi.nl/∼jan

Abstract

We introduce the notion of generative software complexity *to illustrate some of the problems one may run into when trying to tackle a special case of a question recently raised by Brooks.*

The question

In [Bro03], Brooks proposes three great challenges for computer science: *quantification of structural information*, *software estimation*, and *user interface design for computer systems*. Only the first one will concern us here. After recalling the successes of Shannon's information theory, Brooks states

> We have no theory, however, that gives us a metric for the information embodied in structure, especially physical structure. We know that an automobile is a more complex structure than a rowboat. We cannot yet say it is x times more complex, where x is some number.

We indicate some of the problems one may run into when trying to tackle this important question.

Automobiles vs. wooden rowboats

Is an automobile a more complex structure than a rowboat? At first sight, it certainly is, but at what level of structural detail? Is an automobile made of non-biological materials still more complex than a rowboat made of wood? This is less obvious. Wood will probably score high on any structural complexity scale. A procedure or recipe for synthesizing wood out of basic chemical substances might be pretty long, for instance. The only reason we consider wood to be a simple substance is that nature provides it virtually for free. We take its "construction" for granted.

Of course, this is not what Brooks has in mind when he states that an automobile is a more complex structure than a rowboat. He tacitly assumes that the structural complexity of the material the boat is made of does not matter. He is talking about a kind of *generic* rowboat made of a suitable, but otherwise unspecified material. Perfectly good rowboats can be constructed from the same kind of materials cars are made of. Nevertheless, this example indicates that structural complexity has to be measured relative to a set of components or materials whose internal structure does not matter and is ignored. This structural cut-off has to be specified explicitly for the complexity comparison to make sense.

Structural complexity of software

Brooks emphasizes the importance of the quantification of physical structure, but it is no less important and perhaps somewhat easier to contemplate the quantification of software structure. This has the additional advantage of making it a computer science concern (and a fitting subject for SEN) rather than a problem in theoretical physics or chemistry.

Structural complexity of software should not be confused with computational complexity. The latter is concerned with various aspects of the run-time behavior of programs, such as their use of time and memory as a function of input size. Structural complexity of software, on the other hand, is primarily concerned with the structure of the program text as a static object.

Don't we already know how to quantify the structural complexity of software? Certainly, McCabe's cyclomatic and essential complexity measures, both based on the program flowgraph, do just that. Other measures try to capture other aspects [FP96, Chapter 8]. Structural complexity manifests itself in different ways, it seems. This is one of the problems one runs into. It looks as if no single measure is satisfactory for all purposes. Even lines of code (LOC) is a useful indicator of structural complexity.

Generative software complexity

Since there are so many already, let's introduce yet another measure. For the time being we call it *generative software complexity*. It measures the effectiveness of applying program generation techniques to the software in question. The lower the generative complexity, the larger the potential of program generation This measure implicitly underlies software engineering techniques like (imperfect) clone detection [Bak95], program generation [SBnd], and generative programming [Big98, CE00].

To begin with, we fix a general purpose programming language L to write program generators in. In practice, it may be hard to find a suitable language (Lisp comes to mind),

but for the present purpose any Turing-complete language will do.

The programs in whose generative complexity we are interested need not be written in L. Given such a program, its generative complexity is defined as *the length of the shortest program generator (written in L) producing it.*

The length of the generator is measured in number of (lexical) symbols or, less precisely, in LOC. Given the shortest generator G for a program P, the latter can be replaced by a tiny shell script that first runs G and then runs the output of G (which happens to be P). This script is very short, so the total length of G and the shell script combined is for all practical purposes still equal to the length of G.

Generative software complexity does not use a simple measure like LOC directly, but inserts a program generation phase and then applies the simple measure to the generator. The resulting two-stage measure is both shallow and deep:

- It is shallow, because as far as generative complexity is concerned a program is only a string of symbols without meaning.
- It is deep, because there may be deep regularities in the program text whose discovery is highly non-trivial. The shortest generator has to use these regularities to beat other generators. Less attractively, it may also use accidental textual patterns unrelated to the program's intent.

What is generative software complexity?

Generative software complexity is actually the Kolmogorov complexity of software. Kolmogorov complexity is a fundamental notion in information theory (algorithmic information theory), inductive learning, pattern recognition, and other areas [LV97].

The software perspective allowed us to describe generative complexity in software engineering terms and indicate its links with established software engineering practices. The Kolmogorov complexity perspective yields further insights and allows the use of a different terminology:

1. Program generators are compressed programs. Programs with low generative complexity are highly redundant. The shortest generator itself has maximal generative complexity. It cannot be compressed further.

2. After his car vs. rowboat example quoted in the beginning, Brooks added

> Yet we know that the complexity is related to the Shannon information that would be required to specify the structures of the car and the boat.

It can now be seen that this is true for generative software complexity except that it is not Shannon information, but algorithmic information that is involved. In fact, generative software complexity is the amount of algorithmic information needed to specify the program. The executable specification in question is the shortest program generator. Its length (in bits) is the amount of algorithmic information that is needed.

3. Kolmogorov complexity (and hence generative complexity) are *uncomputable*. Asking for the shortest generator is simply too much. We reach a fundamental theoretical limit here.

Concluding remarks and future work

Quantification of structural information is a hard problem, even in the special case of software. Structural complexity manifests itself in different ways, it seems. This leads to different metrics that try to capture different aspects of it.

Generative software complexity (Kolmogorov complexity of software) is implicit in many established software engineering practices, but it is uncomputable. A practically useful and well-defined computable approximation to generative software complexity may exist, however, perhaps based on advanced pattern recognition. This remains to be seen.

An expanded version of this note is in preparation.

References

[Bak95] B. S. Baker. On finding duplication and near-duplication in large software systems. In *Proceedings of the 2nd Working Conference on Reverse Engineering (WCRE '95)*, pages 86–95. IEEE, 1995.

[Big98] T. J. Biggerstaff. A perspective of generative reuse. *Annals of Software Engineering*, 5:169–226, 1998.

[Bro03] F. P. Brooks, Jr. Three great challenges for half-century-old computer science. *Journal of the ACM*, 50(1):25–26, January 2003.

[CE00] K. Czarnecki and U. Eisenecker. *Generative Programming.* Addison-Wesley, 2000.

[FP96] N. E. Fenton and S. L. Pfleeger. *Software Metrics: A Rigorous and Practical Approach.* ITP, second edition, 1996.

[LV97] M. Li and P. M. B. Vitanyi. *An Introduction to Kolmogorov Complexity and its Applications.* Springer-Verlag, second edition, 1997.

[SBnd] Y. Smaragdakis and D. Batory. Application generators. Technical report, Department of Computer Science, University of Texas at Austin, n.d. http://www.cc.gatech.edu/~yannis/generators.pdf.