Science of Computer Programming 97 (2015) 82–85

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

Generative software complexity and software understanding

Jan Heering [1]

IJburglaan 790, 1087 EM Amsterdam, The Netherlands

HIGHLIGHTS

- Introduction of a software complexity metric underlying software generation.
- Generative software complexity is the Kolmogorov complexity of software.
- Discussion of generative complexity as an understandability metric.
- Explanation of the emergence of domain-specific concepts.

ARTICLE INFO

Article history:
Received 26 September 2013
Accepted 5 November 2013
Available online 13 November 2013

Keywords:
Structural complexity
Software generation
Software understanding
Kolmogorov complexity

ABSTRACT

Taking generative software development as our point of departure, we introduce generative software complexity as a measure for quantifying the structural complexity of software. After explaining that it is the same as Kolmogorov complexity, we discuss its merits from the viewpoint of software understanding. The results obtained are in many ways unsatisfactory, but sufficiently intriguing to warrant further work.

Software generation is one of the main themes in Paul's work. The ML/1 macro processor, a now almost forgotten tool for language extension, and Lisp, with its exceptional support for program generation and metaprogramming, were among his early fascinations. Software generation is also at the core of the ASF+SDF Meta-Environment, which offered the unheard of luxury of unrestricted syntax, pioneered lazy generation technology years before Java JIT-compilation became popular, and promoted a unified view of static and dynamic language aspects. The work on ASF+SDF started in 1984 under the title "Generation of Interactive Programming Environments". My cooperation with Paul on ASF+SDF has been a high point in my professional life at CWI.

1. Structural complexity and software understanding

A structural complexity measure may be expected to give an indication of the amount of effort needed to understand the piece of software to which it is applied. Structural or static 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 flow graph, do just that. Other metrics try to capture other aspects

E-mail address: jan.heering1@gmail.com.
[1] Retired from CWI.

0167-6423/$ – see front matter
http://dx.doi.org/10.1016/j.scico.2013.11.017

[5, Chapter 8]. Structural complexity manifests itself in different ways, it seems. This is one of the problems one runs into when trying to find a satisfactory definition. It looks as if no single measure is satisfactory for all purposes. Even lines of code (LOC) is a useful indicator of structural complexity. From the viewpoint of software understanding, these metrics, although useful in software estimation, capture only limited aspects of software complexity. There is ample room to explore other metrics that try to quantify deeper aspects.

What happens if we take the notion of understanding as meant in scientific understanding or software understanding as our point of departure and try to base a complexity measure on it? In this context, understanding means something like *having an explanation in terms of a set of laws or rules*. It is common practice, for instance, for users to try to understand a program's behavior by informal induction of a set of general rules from successive I/O experiments. In this way, users construct a rudimentary theory or model of the program in question. This is the scientific method informally applied to program understanding.

When applying the above-mentioned notion of understanding to a software text rather than to software behavior, there are at least two options. Either the text's meaning is taken into account or it is ignored.

- If the meaning is taken into account, it provides a direct explanation of the text in terms of semantic rules. Although rarely feasible in practice, this may provide full understanding (relative to the meaning of the semantic rules).
- If the meaning of the text is ignored, the only possibility is to provide a partial explanation in terms of recurring textual patterns of a general nature. Ideally, from the viewpoint of software understanding, such patterns would correspond to higher-level semantic notions, facilitating understanding at the semantic level. We introduce a metric to try to quantify the "explanation gain" with respect to the LOC metric that can be obtained.

2. Generative software complexity

Generative software complexity measures the effectiveness of applying program generation techniques to a piece of software. The lower the generative complexity, the larger the potential for program generation. As will be discussed further in Section 4, this measure implicitly underlies software engineering techniques like (imperfect) clone detection [1], program generation [8], and generative programming [2,3].

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

The software S in whose generative complexity we are interested need not be written in L. It may be a real program in some other (or the same) programming language, or a highly declarative specification such as a BNF grammar. Given a piece of software S, its generative complexity is defined as *the length of the shortest program generator* (*written in* L) *producing* S *as its output*.

The length of the generator is measured in number of (lexical) symbols or, less precisely, in LOC. If S is a program P, and given the shortest generator G for 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 and superficial 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 very hard. A minimal generator has to use these regularities to beat other generators.

3. What is generative software complexity?

Generative software complexity is actually the Kolmogorov complexity of a software text, that is, the length of the shortest program generating the text. The choice of programming language for the generator is not critical. It is easy to show that the choice of language affects the complexity value only up to an additive constant. Kolmogorov complexity is a fundamental notion in information theory (algorithmic information theory), scientific understanding, inductive learning, pattern recognition, and other areas [7].

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 information-theoretical terminology:

- 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. There is no understanding to be gained from its structure, which is algorithmically random, without taking the semantics of the language in which it is written into account.