Science of Computer Programming 76 (2011) 1078–1097

Contents lists available at ScienceDirect

Science of Computer Programming

journal homepage: www.elsevier.com/locate/scico

Studying software evolution using artefacts' shared information content☆

Tom Arbuckle*

Computer Science and Information Systems, University of Limerick, Limerick, Ireland

ARTICLE INFO

Article history:
Received 3 August 2010
Received in revised form 7 November 2010
Accepted 9 November 2010
Available online 27 November 2010

Keywords:
Software evolution
Software measurement
Information theory
Kolmogorov complexity
Similarity metric
Information content
CompLearn

ABSTRACT

In order to study software evolution, it is necessary to measure artefacts representative of project releases. If we consider the process of software evolution to be copying with subsequent modification, then, by analogy, placing emphasis on what remains the same between releases will lead to focusing on similarity between artefacts. At the same time, software artefacts – stored digitally as binary strings – are all information. This paper introduces a new method for measuring software evolution in terms of artefacts' shared information content. A similarity value representing the quantity of information shared between artefact pairs is produced using a calculation based on Kolmogorov complexity. Similarity values for releases are then collated over the software's evolution to form a map quantifying change through lack of similarity. The method has general applicability: it can disregard otherwise salient software features such as programming paradigm, language or application domain because it considers software artefacts purely in terms of the mathematically justified concept of information content. Three open-source projects are analysed to show the method's utility. Preliminary experiments on *udev* and *git* verify the measurement of the projects' evolutions. An experiment on *ArgoUML* validates the measured evolution against experimental data from other studies.

1. Introduction

Within software evolution, several broad themes can be identified [1]. Moreover, the term is often used interchangeably with software maintenance [2]. Specifically, therefore, our focus here is on the theme of change of software artefacts in time or across versions. Following the everyday use of the word evolution, we want to examine how software is modified in response to its environment or requests from its users. With our eye on listed challenges in software evolution research [3–5], we want to study how software evolves.

A rich source of evolutionary data comes from mining software repositories [6]. Open-source projects have proliferated largely solving former problems with access to proprietary development information. Given that we can obtain many different kinds of software artefacts, how are their evolutions to be measured?

Software is generally measured using software metrics. This field has a long and venerable heritage [7–10] and a disappointing preponderance of difficulties and disagreements [11–19]. With hundreds of software metrics to choose from, it is difficult to make a convincing case that any one software metric is significantly better. They tend to be designed for particular purposes and need to be calibrated against development context.

An application of information theory, specifically employing a measurement based on the (relative) Kolmogorov complexity, provides a means, disregarding purposes, languages and context, of measuring software and thereby software evolution. Software is, after all, information and that information is representative of the decisions made in its design and

☆ Figures in this paper make use of colour. Obtaining an electronic or colour-printed version of the paper may aid comprehension.

* Tel.: +353 61 23 4284.
E-mail addresses: tom.arbuckle@ieee.org, tom.arbuckle@ul.ie.

0167-6423/$ – see front matter © 2010 Elsevier B.V. All rights reserved.
doi:10.1016/j.scico.2010.11.005

construction. By choosing information theoretic measurement, we attack the hard problem of software measurement with the precision and ineluctability of a mathematically defined concept.

The object of this paper is to continue to reinforce our claims [20–22] by experimentally examining data from open-source projects. After preliminary studies of the projects *udev* [23] and *git* [24], a more detailed exploration of the project *ArgoUML* [25] permits validation of the technique against results obtained by other researchers.

The structure of the remainder of this paper is as follows. First, we clarify our terminology and relate shared information measurement to the software engineering literature on measurement and comparison. We state the thesis of the paper and the approach to be followed in validating it. Then, a self-contained section provides only the background theory necessary to understand the experiments. Next we describe the experimental approach. The steps to be followed, interpretation of results, and threats to validity are detailed. We conduct two preliminary studies. There follows a third, comparative, experiment in which the results obtained are validated against those of other researchers. Related work is outlined. The conclusions and future work are presented. Finally, appendices on certain technical aspects of the theory are provided, together with the bibliography.

2. Theme and thesis

2.1. What is software evolution?

We define software evolution as the way in which software artefacts change between versions. A version need not represent a full release but could simply represent the current development status. Versions need not be chronologically ordered. Provided the artefacts are representative, we do not restrict their type. In practice, we often consider source code, representations of structure, representations of behaviour or the machine instructions themselves.

There is an extensive literature on how software evolves in this sense. Mens and Demeyer's paper concerning software evolution metrics [26] provides key references. In addition, Fernandez-Ramil et al. [27] have written a review starting from the early work of Belady and Lehman and going on to discuss studies of evolution of open-source systems. Israeli and Feitelson's study of Linux kernel evolution [28] provides a recent notable addition.

2.2. Software measurement = (software) metrics?

Evolution implies change and in order to quantify change, we need a measurement method. Software is traditionally measured using software metrics, more properly called software measures [29].

Early methods of measuring software include counting instructions [30] or lines of code (LoC) [31] with LoC being suggested as a baseline software measurement as late as 1983 by Basili and Hutchens [32]. Further early publications on software metrics include Rubey and Hartwick [33]. McCabe's cyclomatic complexity [7], and Halstead's software science [8]. Reading early reviews of the field, such as Perlis et al. [9] or Cook [34], it is plain that, on the one hand, the need for means of measuring software has been clearly recognised, and, on the other hand, the field has already run into difficulties and been the topic of intense debate. Indeed, as we have already mentioned, metrics are also the subject of subsequent critical papers [11–14]. More recent metrics including those by Chidamber and Kemerer [10], the metrics by Lorenz and Kidd [35] and the MOOD set of metrics [36], focus on the measurement of object-oriented code. While Chidamber and Kemerer's metric suite's popularity means that it has almost become a *de facto* standard, it has not escaped extensive criticism [15–17]. The MOOD metrics have also been criticised [18] although other authors have found them to be useful [37]. The Lorenz and Kidd metrics are similarly the subject of claim and counter-claim [38,39]. It is clear that the subject of the measurement of software is both difficult and contentious. As Abran et al. state in their 2003 paper [19]

> This is a clear indication that, when looked at from an engineering perspective, measurement in software engineering is far from being mature and that it constitutes a fairly weak engineering foundation for the field of software engineering.

One area of software measurement that has gained acceptance is the area of software measurement for project estimation and management. From early work by Putnam [40] and Albrecht [41], function point measurement has progressed to become a necessary part of software process improvements initiatives such as CMM or CMMI. Boehm et al. and Jones describe two current leading methods [42,43].

Finally, attempts to employ information theory to measure software have, almost without exception, involved the use of (Shannon [44]) entropy. Starting from Campbell [45] and Hellerman [46], two recent examples are Sarkar et al. [47] or Anan et al. [48]. We will not employ entropy. See Appendix A for more details.

2.3. Edit distances and beyond

There are alternatives to using software metrics and comparing the values they produce on different artefacts. During coding, a developer comparing two files will commonly use a file differencing tool, such as the UNIX command *diff* to show the points at which the files differ. Such tools employ the Myers algorithm [49] (or a variant [50]) to create a graph