Applied Soft Computing xxx (2016) xxx–xxx

Contents lists available at ScienceDirect

Applied Soft Computing

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

On deterministic chaos in software reliability growth models

Q1 O. Yazdanbakhsh, S. Dick (Dr.) (Associate Professor)*, I. Reay, E. Mace

Dept. of Electrical & Computer Engineering, University of Alberta, Edmonton, AB, Canada

ARTICLE INFO

Article history:
Received 1 December 2015
Received in revised form 31 May 2016
Accepted 2 August 2016
Available online xxx

Keywords:
Software reliability
Chaos theory
Time series analysis
Machine learning
Forecasting

ABSTRACT

Software reliability growth models attempt to forecast the future reliability of a software system, based on observations of the historical occurrences of failures. This allows management to estimate the failure rate of the system in field use, and to set release criteria based on these forecasts. However, the current software reliability growth models have never proven to be accurate enough for widespread industry use. One possible reason is that the model forms themselves may not accurately capture the underlying process of fault injection in software; it has been suggested that fault injection is better modeled as a chaotic process rather than a random one. This possibility, while intriguing, has not yet been evaluated in large-scale, modern software reliability growth datasets.

We report on an analysis of four software reliability growth datasets, including ones drawn from the Android and Mozilla open-source software communities. These are the four largest software reliability growth datasets we are aware of in the public domain, ranging from 1200 to over 86,000 entries. We employ the methods of nonlinear time series analysis to test for chaotic behavior in these time series; we find that three of the four do show evidence of such behavior (specifically, a multifractal attractor). Finally, we compare a deterministic time series forecasting algorithm against a statistical one on both datasets, to evaluate whether exploiting the apparent chaotic behavior might lead to more accurate reliability forecasts.

1. Introduction

The smartphone revolution has embedded software into the daily lives and routines of hundreds of millions of ordinary people. We text and message instantly with friends and acquaintances across the street or across the country. We pull out our phones to search for information at the drop of a hat. Our schedules, emails, and contact lists are literally at our fingertips. Software monitors and controls all major industrial plants, the power grid, sewer and water systems, and public transportation. Government documents are now posted in official online repositories for all citizens to view. News articles, streaming media, online games, etc. pour nearly a zettabyte of data across the Internet each year [80]. Software is intimately woven through every aspect of our lives, and so software failures pose an immediate and critical danger to life, limb and property. It is thus disconcerting that software is more failure-prone than any other engineered product [33]. Software failures likely cost the U.S. economy over $78 billion per year [40]. Improving the reliability of software systems is thus one of the most crucial technical challenges of the 21st century.

While software quality is generally poor, this is not merely a case of shoddy work, but a testament to the sheer intellectual difficulty of developing large software systems. With codebases stretching to hundreds of millions of lines long, and more than 10^{20} possible states, software systems are by orders of magnitude the most complex creations mankind has ever attempted to build [20]. By way of comparison, the human brain contains on the order of 60 trillion connections between neurons [73]; thus, it is reasonable to say that software developers are trying to build something orders of magnitude more complex than their own brains. In the face of this complexity, defect-free software is an unachievable goal; instead, we must ensure that the number and severity of the defects remaining when a product is shipped are acceptably low. It does not matter if a few pixels on an in-car entertainment display are the wrong colour; it matters a great deal if the navigation system directs drivers over a cliff.

Software quality assurance almost uniformly follows the develop-and-test paradigm. Code is written, and then its behavior is compared to its specification by executing a number of inputs that should result in known outputs. If the actual and expected outputs match, that test passes; if not, a bug has been found and must be fixed. Thus, the main question for project managers is how to

* Corresponding author.
E-mail address: dick@ece.ualberta.ca (S. Dick).

http://dx.doi.org/10.1016/j.asoc.2016.08.006
1568-4946/

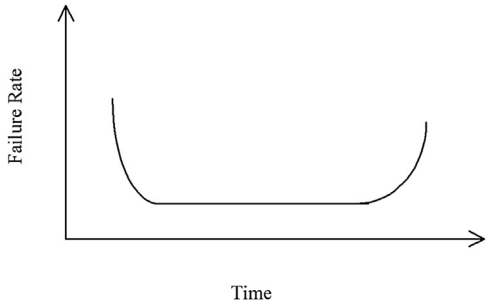

Fig. 1. Failure Rates over Time [41].

determine *how much* testing is enough for the software to reach an acceptable level of quality. From a management perspective, this means that the expected future cost of maintenance and liability (responding to failures in the field) is acceptably low. When the software reaches this point, it is ready to be released. Ideally, organizations should use the development and testing history of the software system to decide when the release point has been reached. Heuristic criteria that are commonly used include requiring that the number of open minor and moderate bugs be below some threshold, that no serious or safety-critical issues are open, that the rate at which bugs are discovered and their severity are both clearly decreasing, etc. [11]. A more optimal approach would set a threshold of expected future costs below which the software may be released. In order to do this, the probability of a failure in the field over time needs to be estimated; in other words, the future reliability of the system must be forecast. In reliability engineering, this forecast is produced by a *reliability growth model* [41].

Reliability growth models for physical systems are substantially different from those for non-physical systems such as software. In a physical system, failure rates over time generally follow the "bathtub curve" depicted in Fig. 1. After an initial period of high failure rates due to residual design defects (the "infant mortality" stage), the system reaches regular operation, during which time failures are generally due to random external events (hence a constant, low rate of failures). As the system's components wear out, the failure rate again increases, until the system reaches its end of life [41]. The key difference for software is that physical systems are subject to wear and random defects in material components. By contrast, *all* software failures are due to residual design defects; in a very real sense, software never exits the infant mortality stage. Thus, instead of the exponential-class models often used for hardware reliability (e.g. the Weibull distribution [41]) counting models such as Non-Homogenous Poisson Processes (NHPPs) are frequently used for Software Reliability Growth Models (SRGMs), e.g. [53,70].

There has recently been interest in a different class of SRGMs, based on the notion that fault injection in software is a *chaotic* process, rather than a random one. Studies in [12,95] and others have examined SRGM datasets, and found signatures of chaotic behavior in them (chiefly by treating the SRGM dataset as a time series from a dynamic system, and determining a fractal dimension for the state-space attactor). The hypothesis of chaotic behavior is intriguing, but the studies above share a common weakness: publicly-available software reliability growth datasets are small, usually consisting of only a few hundred observations. There may be thousands of failures tracked by these datasets, but they have usually been summarized into counts of failures per time interval. The resulting time series are now usually considered too small to reliably test for chaotic behavior (although a recent advance allowing modeling of even short chaotic time series is reported in [71]). What is needed for a definitive test of the chaotic hypothesis are large SRGM datasets drawn from modern large-scale software systems. This preserves the fine-grained structure of the data, and provides enough data for a reliable test. [12] performed their analysis on the largest inter-failure time datasets that were then available, which contained no more than 2000 observations (the Musa dataset [46] is an inter-failure dataset, but only contains 136 observations). This is barely adequate for testing for the simplest form of chaotic behavior (i.e. a monofractal state space attractor); an order of magnitude more data will be needed to reliably test for multifractal behavior. A further necessary test of the chaotic hypothesis is to determine if it leads to a superior model than random behavior. In our view, this means comparing nonlinear deterministic models against probabilistic ones in a forecasting experiment.

Our goal in the current paper is to employ four large-scale software reliability datasets to test for chaotic behavior, and compare probabilistic and deterministic forecasting algorithms on these datasets. We introduce two new SRGM interfailure time datasets, one derived from bug reports and change records for the Android open-source operating system (this is the 2012 MSR Challenge dataset[1]), and a second from the defect-tracking database for the Mozilla open-source Web browser. The Android dataset is an order of magnitude larger than any existing SRGM dataset, at over 20,000 observations; the Mozilla dataset contains more than 86,000 entries. We test for chaotic behavior in these datasets, as well as the two datasets from [12] by estimating the fractal dimension of the underlying attractor. We furthermore compute the multifractal spectrum of these datasets, an analysis which has not previously been attempted. We find that both new datasets, as well as one of the datasets used in [12], have multifractal attractors and are thus chaotic; we discuss how this complex geometry may have arisen in the context of normal software engineering practice. In forecasting experiments, we find that all three datasets also exhibit long-range dependencies, and that fractional ARIMA models and radial basis function networks are equally effective in forecasting them.

Our contributions to the literature are as follows: 1) we develop and analyze two new SRGM datasets, which are an order of magnitude larger than any currently available in the public domain; 2) we determine that the state-space attractors for both of these datasets, as well as an existing one, have a multifractal geometry; 3) we show that stochastic (fractional ARIMA) and deterministic (radial basis function network) models are equally effective in modeling these datasets.

The remainder of this paper is organized as follows. In Section 2, we review essential background and related work, ultimately developing our research hypotheses. We describe our methodology in Section 3, and discuss the creation of our datasets and our initial processing of them in Section 4. We present our extraction of fractal dimensions and multifractal spectra in Section 5, and our predictive modeling experiments in Section 6. We discuss and interpret the totality of our results in Section 7, and close with a summary and discussion of future work in Section 8.

2. Related work

The earliest bespoke SRGMs were the Jelinski-Moranda de-eutrophication model [32], and Schneidewind's Non-Homogenous Poisson Process (NHPP) model [70]. Musa's basic execution model [52], the Yamada S-shaped model [89] and Musa & Okumoto's logarithmic Poisson model [53] were developed a decade later. All of these models are variations on the NHPP concept, which is a Poisson process whose mean value is time-varying:

$$P\{N(t) = k\} = \frac{(m(t))^k}{k!} e^{-m(t)} \quad (1)$$

[1] http://2012.msrconf.org/challenge.php.

Please cite this article in press as: O. Yazdanbakhsh, et al., On deterministic chaos in software reliability growth models, Appl. Soft Comput. J. (2016), http://dx.doi.org/10.1016/j.asoc.2016.08.006