# Software-Reliability Modeling: The Case for Deterministic Behavior

Scott Dick, *Member, IEEE*, Cindy L. Bethel, and Abraham Kandel, *Fellow, IEEE*

***Abstract*—Software-reliability models (SRMs) are used for the assessment and improvement of reliability in software systems. These models are normally based on stochastic processes, with the nonhomogeneous Poisson process being one of the most prominent model forms. An underlying assumption of these models is that software failures occur randomly in time. This assumption has never been quantitatively tested. Our contribution in this paper is to conduct an experimental investigation that contrasts random processes with nonlinear deterministic processes as a model for software failures. We study two sets of real-world software-reliability data using the techniques of chaotic time-series analysis. We have found that both appear to arise from a deterministic process, rather than a stochastic process, and that both show some evidence of chaotic dynamics. In addition, we have conducted a series of k-steps-ahead forecasting experiments in the datasets, pitting a number of well-known stochastic SRMs against radial basis function networks (RBFNs), which are deterministic in nature. The out-of-sample prediction results from the RBFNs showed an improvement of roughly 25% over the best of the stochastic models, for both of our datasets. Finally, we propose a causal model to explain these results, which hypothesizes that faults in a program are distributed over a fractal subset of the program's input space.**

***Index Terms*—Chaos theory, fractal sets, soft computing, software quality, software reliability, time-series analysis.**

## I. Introduction

SOFTWARE reliability is a critical technological challenge for the 21st century; as software plays a greater and greater role in our society, the reliability of that software becomes a key concern. Lives and property already depend on a variety of software systems; when these systems fail, either or both can be at risk. The Therac-25, Ariane-5, and Mars Climate Observer system failures are well known and widely documented examples of the consequences of software failures. Obviously, software developers would like to engineer reliability into their systems, in much the same way that engineers in other disciplines do. However, the sheer complexity of software makes this extremely difficult. As far back as 1995, a large software system might have $10^{20}$ states [11]; since then, software has only gotten more complex. Systems comprising tens of millions of lines of source code are now widely distributed, with the Windows XP and OpenOffice systems being two prominent examples.

Reliability, in the general engineering sense, is "the probability that a given component or system within a specified environment will operate correctly for a specified period of time." In general, the probability of correct operation is inversely related to the length of time specified; the longer a system operates, the greater the chance of failure. The relationship between the component or system and its environment is more complex; placing the component or system in a different environment may increase or decrease the chances of failure [22]. Software reliability is defined as "the probability that a given software system within a specified environment will operate correctly for a specified period of time." A software failure occurs when the observed behavior of a software system departs from its specified behavior. Software failures are ultimately the result of faults in a program, which are the human mistakes made during the construction of the system. Faults are triggered by an input to the system, thereby causing an error—an internal departure of the system from its correct behavior. An error becomes a failure when it propagates all the way to the system output [11].

Software-reliability models (SRMs) are used by developers to answer two basic questions about their software systems: When will the software be reliable enough to ship, and what will its expected reliability be at that time [26]? A basic assumption in SRMs is that software failures are the result of some stochastic process, having an unknown probability distribution. Each SRM specifies some reasonable form for this distribution and is then fitted to the reliability data for a given project. The question we ask in this paper is why are software failures modeled as stochastic processes? After all, no developer rolls a set of dice and says, "Aha! Time to make a mistake!" Unlike failures in hardware systems where the random occurrence of material defects is unavoidable, software failures are the result of human mistakes, which do not appear to be random in nature. We defer a discussion of the exact meaning of the word random to Section III.

An alternative explanation for the unpredictable nature of software failures is that they might be chaotic in nature rather than random. To quote from the preface to Kantz and Schreiber [20]; "Deterministic chaos offers a striking explanation for irregular behavior and anomalies in systems which do not seem to be inherently stochastic... The framework of deterministic chaos constitutes a new approach to the analysis of irregular time series. Traditionally, nonperiodic signals have

Manuscript received October 31, 2002; revised September 16, 2003, November 18, 2004, and September 6, 2005. This work was supported in part by the Natural Sciences and Engineering Research Council of Canada under Grants PGSB222631-1999 and G121210906, and in part by the National Institute for Systems Test and Productivity at the University of South Florida under the U.S. Space and Naval Warfare Systems Command Grant N00039-01-1-2248. This paper was recommended by Associate Editor H. Pham.

S. Dick is with the Department of Electrical and Computer Engineering, University of Alberta, Edmonton, AB T6G 2V4, Canada (e-mail: dick@ece.ualberta.ca).

C. L. Bethel and A. Kandel are with the Department of Computer Science and Engineering, University of South Florida, Tampa, FL 33620 USA (e-mail: kandel@csee.usf.edu).

Digital Object Identifier 10.1109/TSMCA.2006.886364

been modeled by linear stochastic processes. But, even very simple chaotic dynamical systems can exhibit strongly irregular time evolution without random inputs. Chaos theory offers completely new concepts and algorithms for time series analysis. . .” In this paper, we will approach software-reliability modeling by investigating two sets of software-reliability data using the methods of nonlinear time-series analysis. Our results do, indeed, indicate that a deterministic process, rather than a stochastic one, is most likely at work in these datasets. Furthermore, there is some evidence of chaotic dynamics in these datasets, although this latter is not conclusive. We then turn our attention to validating our claims through predictive modeling; the true test of a new model is whether its predictions are an improvement over the previous ones. Our experimental design is a leave-$k$-elements-out forecasting experiment, and our performance measure is the root-mean-square (rms) prediction error on the $k$ test points. We use radial basis function networks (RBFNs)—a form of deterministic model highly recommended in [20]—to model both datasets, and we compare the rms prediction error in these models with the rms error from several well-known SRMs. Our results are striking; the RBFN models showed an improvement of roughly 25% over the stochastic models. These results support our contention that deterministic chaos is a viable alternative to stochastic processes in software-reliability modeling.

The remainder of this paper is organized as follows. In Section II, we review the literature on software-reliability engineering and software-reliability modeling. In Section III, we discuss the difference between randomness and deterministic chaos. This discussion is in large measure philosophical, but the practical implications are quite significant. Section IV briefly reviews the process of chaotic time-series analysis, and this process is then used in Section V to analyze our two datasets. In Section VI, we present the results of our predictive modeling experiments for these datasets, and we discuss the significance and implications of these results in Section VII. We close with a summary and discussion of future work in Section VIII.

## II. Software-Reliability Engineering

Unlike hardware systems which are physical objects in the real world, software systems are logical constructs having no physical existence. This leads to a number of important differences between hardware and software systems. For instance, software is plainly not subject to any kind of physical aging. However, as a system is maintained over a number of years, faults are injected by the maintainers. Over time, this leads to a breakdown of the system’s conceptual integrity and thus could be considered as a kind of “logical aging” [12]. On the other hand, there are also areas in which there is a considerable similarity between hardware and software systems, such as in their response to an increased load. In hardware systems, it is well known that the reliability of a system decreases (possibly in a nonlinear fashion) when the system is subjected to an increased load [22]. In this context, the work of Iyer and Rossetti [17] is very important, as it establishes a similar behavior for software systems. They studied the performance of an IBM 3081 at the Stanford Linear Accelerator Center and found that the number of failures observed was correlated with the volume of interactive processing (paging rate, operating system CPU time, etc.), but was not correlated with the overall CPU utilization. This indicates that “loads” for a software system are the interactive operations and not compute-bound processes.

Software-reliability engineering is based on the collection of reliability growth data only, as opposed to the collection of both reliability growth and life data in hardware reliability. This, of course, is due to the logical nature of the software; life testing is an attempt to estimate the mean and variance of the lifetime of a system in operation, by running many copies until failure under identical conditions. Clearly, since there is no variability between two copies of a software system, there is no need to conduct life testing. Reliability growth data, on the other hand, consist of the times between successive failures of a system under test; this corresponds with the usual debug-and-test cycle used in software development. Once these data are collected, we can attempt to fit an SRM to the model and thereby obtain the reliability function, failure rate, and mean time to failure for the software system. Researchers have been studying SRMs since the early 1970s; some of the early models include the Jelinski–Moranda de-eutrophication model [18], and Schneidewind’s 1975 nonhomogeneous Poisson process (NHPP) model [38]. The Jelinski–Moranda model was popular for much of the 1970s, until it was supplanted by Musa’s basic execution model [29] and later the logarithmic Poisson model developed by Musa and Okumoto [31]. All of these models are examples of the NHPP, which is a Poisson process having a time-varying mean value function. An NHPP is a counting process $N(t)$ of the form

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

where $N(t)$ is the number of events observed by time $t$ and $m(t)$ is the mean value function. The NHPP is a very common framework for SRMs; development of new NHPP models continues to this day, with some recent examples found in [15], [16], [33], and [46]. Another main framework for SRMs is Bayesian inference; the first of these was the Littlewood–Verrall model in [25], with the studies in [34], [35], and [40] being more recent examples. For an extensive overview of SRMs, we direct the reader to [9], [30], [32], or [42].

A few papers have used time-series analysis in the construction of SRMs, by treating reliability growth data as an autoregressive (AR) process. In [41], a time series of software-reliability data was assumed to arise from a power-law process

$$T_i = \delta_i \cdot T_{i-1}^{\theta(i)} \tag{2}$$

where $T_i$ is the time to the $i$th failure, $\theta(i)$ is an unknown constant, and $\delta_i$ is a random coefficient. By assuming that the $T_i$s and $\delta_i$s are lognormally distributed, we can take the logarithm of both sides and obtain a first-order AR process. A more complex time-series model based on an AR integrated moving average process is presented in [21]. This paper also

stands out because a small number of software complexity metrics are integrated into the model along with reliability growth data.

To our knowledge, the idea of using chaos theory in software-reliability modeling has only been looked at once by Zou and Li in [48]. Their argument for using chaos theory is similar to our discussion in Section III below that a software system itself is a deterministic mapping from inputs to outputs and that the test cases selected during software testing are chosen based on a test plan, rather than being random samples from a population. The model Zou and Li propose is a locally linear approximation. This model performs reasonably well on in a one-step-ahead prediction experiment. However, as will be discussed in Section III, locally linear approximations swiftly break down when forecasting chaotic systems more than a single time step ahead. In addition, there are methodological problems in [48], which we will discuss in Section IV after reviewing nonlinear time-series analysis.

## III. Randomness Versus Chaos in Software-Reliability Modeling

The physical world around us is (at least at the macroscale) a deterministic universe; meaning that if one can exactly reproduce all of the conditions surrounding some particular event, then you will be able to exactly reproduce the event itself. The catch is in the term exactly; the universe or any other complex system is more complicated than a human mind can grasp and more precise than even our best tools can measure, so that perfect knowledge of all the conditions surrounding an event is impossible. Thus, all models of complex systems in the real world necessarily incorporate some degree of uncertainty. Software systems, in contrast, are logical in nature and thus can be measured perfectly. However, uncertainty is still a major factor in software-reliability modeling due to the sheer complexity of modern software systems. This is the motivation for using stochastic models of software reliability; stochastic processes represent one mechanism for incorporating uncertainty into a mathematical model. However, they are only one mechanism among several for this purpose.

Stochastic processes and probability theory were developed to model a particular form of uncertainty commonly known as "randomness." This is the species of uncertainty concerned with events that follow the law of large numbers. This concept is based on the idea of repeatedly sampling an uncertain phenomenon or population. For an individual sample, one cannot be certain what the outcome will be. However, if the law of large numbers holds, then we can predict what the overall pattern of outcomes over a large number of samples will be. According to the law of large numbers, in the limit of an infinite number of samples, the sample mean will converge to the theoretical mean of the population [37]. This is the form of uncertainty that is assumed in all of probability theory and in stochastic processes. However, this is not the only form of uncertainty that exists. One different form of uncertainty lies in the imprecision at the heart of human language and perception. Consider the idea of a "tall" man; what would this man's height be in meters? Even when we confine ourselves to a single geographic area, we do not obtain a precise answer to this question. The uncertainty present in this situation arises from a vague description of a physical phenomenon; it is a form of uncertainty that we humans encounter constantly, but it is distinct from randomness. One very successful mathematical representation of this uncertainty is the fuzzy set theory [27]. Generally speaking, there are numerous forms of uncertainty; each of which can be matched with a modeling technique that is best suited to quantify it.

The phenomenon of irregularity is also considered a form of uncertainty. Irregularity refers to the occurrence of rare and unpredictable phenomena within a system, such as intermittency or bifurcations. Irregularity arises from nonlinear dynamics, in which a slight change in the system's initial conditions can lead to huge changes in behavior. Chaotic dynamics are a particular example; one of the real challenges in analyzing chaotic systems is that, even though a chaotic system may be driven by an analytic equation, no amount of empirical observation will allow the observer to deduce that equation. This is because, in a chaotic system, two state trajectories that are infinitesimally close in state space will diverge at an exponential rate through time (assuming a dissipative system). Thus, even the slightest measurement error will be magnified exponentially through time. Irregularity thus represents a fundamental limit on how predictable a system is. Even an infinitely large sample from a chaotic system does not allow one to deduce the underlying chaotic equation [20]. The best that can currently be done in analyzing chaotic time series is to employ nonlinear models to make predictions for a short time horizon. Locally linear models can also be applied; however, the time horizon over which useful predictions can be made is even shorter than for nonlinear models [20]. The techniques of nonlinear time-series analysis are currently the best mechanism for mathematically quantifying irregularity, just as probability theory is the current best mechanism for quantifying randomness.

In light of this discussion, let us now consider software failures. To date, the software-reliability-modeling literature has simply assumed that failures are random events, with no quantitative evidence to back up this assertion. Previous authors have, at most, offered a few qualitative statements to justify their treatment of time-to-failure or failures per interval as a random variable. The arguments used include the statement by Musa that "since the number of failures occurring in infinite time is dependent on the specific execution history of the program, it is best viewed as a random variable. . ." [31] (interestingly, Musa is saying here that there will not be a convergence to a theoretical mean), or simply the assumption that "the life lengths of the software at each stage of development are random variables. . ." [27]. A more specific assertion, due again to Musa, is that, while a programmer may not make errors randomly, the location of errors in the code is random and the test case inputs applied to the software are random [29]. The latter point has been extensively criticized. Littlewood [24], Schneidewind [38], Cai *et al.* [3], and Zou and Li [48] have all pointed out that test cases are not randomly selected, but chosen based on a test plan. As for the assertion that the location of faults in code is random, the well-known tendency for faults to cluster in specific modules in a program can be viewed as

evidence that the distribution of faults in a program is also not random. Furthermore, Singpurwalla and Wilson [42] admit that the location of faults within a software system does not appear to be random. They argue that ". . .the different input types can be envisaged as arriving to the software randomly, leading to the detection of errors in a random way. Therefore, although software failure may not be generated stochastically, it may be detected in such a manner." This again is the argument that test case inputs to a software system are randomly generated. A further argument against faults being randomly distributed in code is Brooks' decades-old observation that the conceptual integrity of a system is an essential determinant of its reliability [1]. It is an accepted fact that human mistakes, usually resulting from a breakdown in the developer's conceptual grasp of a system (be it misunderstood/incomplete requirements, a poor mapping from requirements to design, from design to implementation, etc.), lie at the bottom of all software failures. These events are infrequent, unpredictable occurrences that do not seem to follow any probability distribution. Based on the foregoing, we propose that software failures appear to be irregular events, rather than random events, and that the techniques of chaos theory may therefore be used in software-reliability modeling. In the next section, we will provide a brief overview of these techniques.

## IV. Nonlinear Time-Series Analysis

Nonlinear time-series analysis (or equivalently chaotic time-series analysis) is used to extract invariant features from systems that exhibit deterministic chaos. These invariants are drawn from chaos theory and include Lyapunov exponents (which measure the average rate of divergence between initially close trajectories over a manifold) and fractal dimensions. Two important points must be acknowledged at the outset. First, nonlinear time-series analysis is not as completely developed as its linear counterpart and, in some ways, is likely to remain so for a long time to come. Recall from the discussion in Section III that, even in the limit of an infinite amount of empirical observation, it is impossible to deduce the exact form of a chaotic equation simply by observation. Hence, we cannot fit a "chaotic" model to a time series in the way that a linear model can be fitted to a linear time series. What is usually done is to use nonparametric techniques (such as RBFNs) to approximate the dynamics of the chaotic system and produce forecasts of the time series for a short time horizon. Second, large amounts of data, with a miniscule amount of noise, are required in order to extract chaotic invariants. Datasets on the order of 10 000 elements or more are usually used in laboratory experiments on chaotic time series, while even the most robust algorithms for chaotic invariants cannot tolerate a noise amplitude of more than 2%–3% of the overall signal. Considering that neither of our datasets are larger than 2000 elements and that they are contaminated with significant discretization noise, we must acknowledge that these datasets are not of the best quality for nonlinear analysis. However, they represent the very largest datasets available in the open software-reliability literature; most other such datasets contain a few hundred data points or less, far too small for nonlinear analysis [20].

A time series is a sequence of scalar measurements taken over time from some interesting system. These scalar measurements are a complex one-dimensional (1-D) projection of the true unobserved state variables of the system. In order to analyze a system solely from a time series, we must first reconstruct the state space of the system. While we cannot uniquely identify the original state space, we can construct an equivalent state space (in the sense that the two are related by a smooth invertible mapping) using the method of delay embeddings. Let a time series of $k$ measurements be denoted by $x_1, x_2, \ldots, x_k$. A delay embedding of this time series is a sequence of delay vectors $B_n = (x_{n-(m-1)\nu}, x_{n-(m-2)\nu}, \ldots, x_{n-\nu}, x_n)$. This $m$-dimensional vector is formed by combining successive elements of the time series together, with the time lag $\nu$ allowing us to select every consecutive element, every second element, and so on. The parameters $m$ and $\nu$ must be chosen for each time series. There is no definitive algorithm that provides values for both $m$ and $\nu$ on an arbitrary dataset, but there are some heuristics available. The embedding dimension $m$ does undergo a qualitative change from values of $m$ that are less than the true dimensionality for the state space to those that are greater than or equal this true value. Values of $m$ that are too small leave unresolved projections of the state variables, producing false neighbors. Values of $m$ which are large enough remove these false neighbors, and so a search for false neighbors is a powerful technique for finding a reasonable value of $m$. As for the time lag $\nu$, every choice of $\nu$ is mathematically equivalent to every other choice. However, a good selection of $\nu$ enhances nonlinear analysis, and a poor choice hinders it. Small values of $\nu$ enhance correlations, concentrating all delay vectors on the diagonal. A large value of $\nu$ makes successive delay vectors almost independent, causing them to fill a cloud in the phase portrait. The best strategy is to select a promising range for $\nu$ and then construct 2-D phase portraits for each delay. Then, the phase portraits are manually examined, in the hope of finding possible deterministic structure (i.e., trajectories or major voids in the phase portrait). We want to use a value for $\nu$ that produces the largest possible deterministic structures, because noise in the time series will deny us access to infinitesimal length scales [20].

The algorithms for nonlinear analysis assume that the time series comes from a stationary deterministic process, at a sufficiently high sampling rate. If this assumption is violated, then the algorithms will produce spurious results. A check for stationarity in nonlinear time series is thus required. Linear statistics such as autocorrelations do not help here; a parameter drift in a chaotic system might not alter its linear statistics but only its nonlinear statistics. Instead, a technique called the space-time separation plot is used [20], [36]. This technique relates the temporal difference between two points and their spatial distance by finding the fraction of points lying within a specified radius of each other for a specified temporal difference between them. Curves of constant probability for one point to be within a neighborhood of another point are plotted. If these curves saturate in a plateau or stable oscillation, we can conclude that the time series appears to be stationary and thus a valid target for nonlinear analysis. In addition, the space-time separation plot allows us to determine the extent of temporal

correlations between elements of a time series. Temporal correlations are natural in a time series drawn from a physical system; points very close to one another in time should also be spatially close. Temporal correlations are now recognized as a major problem in estimating the geometry of a chaotic attractor; points which are temporally correlated may not be geometrically related and yet would be treated as neighbors. Thus, when estimating a fractal dimension, you must exclude points that occur too closely together in time. The space-time separation plot allows us to estimate the size of that temporal neighborhood [20].

In addition to the check for stationarity, a check for determinism is also required. Theoretically, the algorithms for computing chaotic invariants from a time series will also provide a clear distinction between deterministic chaos and a linear stochastic process; however, those theoretical results are based on the assumption of an infinite series of noiseless observations. In reality, one can appear to extract chaotic invariants even from purely random time series because finite time series can appear to possess "deterministic" structures that would be revealed as purely random fluctuations over an infinite time. Furthermore, most real-world physical systems seem to be a blend of linear stochastic and deterministic elements, in varying degrees of strength. Thus, in the literature on deterministic chaos, the method used to distinguish between a predominantly random and a predominantly deterministic time series is a type of statistical hypothesis test called the method of surrogate data [44]. This test is intended to reveal whether a time series exhibits a predominantly random or deterministic behavior. The null hypothesis is that the time series is best explained as a linear stochastic process (such as a linear Gaussian process); if our test statistic rejects that hypothesis with at least 95% confidence, there is a basis for claiming that the time series is predominantly deterministic in nature. We first construct a group of time series that is random in nature but possesses the same linear characteristics as the original time series and then subject each of these "surrogate" time series, as well as the original signal, to a test that discriminates between random and deterministic behaviors (examples include the error from a nonlinear prediction algorithm or the time reversal asymmetry statistic). If the original time series has a significantly different value from all the other random series, we reject the null hypothesis. "Significant difference" has to be measured using a ranking approach, rather than the tails of a probability distribution; the exact distribution of values from these test statistics are unknown and could be very far from the normal distribution. Thus, the null hypothesis is rejected if the test statistic for the original signal is maximal or minimal with respect to the values of the surrogates, and the significance of the test is determined by the number of surrogate datasets created. Note that, while we are trying to measure deterministic behavior in a time series, we do so by constructing a population of random time series. This population is what is used in the statistical hypothesis test. This approach is the current best practice in nonlinear time-series analysis [20].

The method of surrogate data was developed to test the most general version of the null hypothesis available in the literature: that the original dataset was generated by a linear Gaussian process, distorted by an unknown nonlinear observation function. Each surrogate dataset is an elaborate shuffle of the original dataset, using a linear Gaussian series as the basic template. This ensures that simple linear statistics such as the mean and variance of the time series are not changed. It is also possible to preserve the power spectrum from the original time series in the surrogate series, ensuring that the autocorrelations within the surrogates match the original signal [20].

Any realistic time series will be contaminated by noise. This noise is a very serious problem for nonlinear time-series analysis. Even the most robust algorithm for computing a chaotic invariant, the Grassberger correlation dimension, cannot tolerate a noise amplitude in excess of 2%–3% of the total amplitude of the time series. Thus, the use of a nonlinear noise-reduction algorithm is an important step in nonlinear analysis. One such scheme, the locally constant projective scheme, has been described in [20] and implemented in [14]. Assume that a point $X$ in an $m$-dimensional delay embedding has $k$ neighbors within a radius of $\varepsilon$. We predict that the next point along the system trajectory passing through $X$ will be the mean of the one-step-ahead evolutions of all $k$ neighbors. This noise-reduction technique takes advantage of continuity; even in a chaotic system, two nearby state trajectories cannot diverge at more than an exponential rate through time. This algorithm has been found to be quite robust and has been applied to a number of systems [20].

If a system is chaotic, then its attractor in phase space has a fractal geometry [20]. Fractal sets exhibit self-similarity and have a complex structure at all length scales. In fact, fractal sets are characterized by a Hausdorff dimension strictly less than the topological dimension of the set [8] and frequently by a noninteger value for the Hausdorff dimension. The attractor dimension is an invariant quantity for a chaotic system, although it does not uniquely determine the system. We use the well-known Grassberger correlation dimension algorithm to estimate the attractor dimension in our two datasets. The correlation sum is given by

$$C(\varepsilon) = \frac{2}{N(N-1)} \sum_{i=1}^{N} \sum_{j=1}^{N} \Theta\left(\varepsilon - \|x_i - x_j\|\right) \qquad (3)$$

where $N$ is the number of delay vectors in the time series, $\varepsilon$ is a neighborhood, $\Theta$ represents the Heaviside step function, and $x_i$ and $x_j$ are delay vectors with $i \neq j$. The correlation sum counts the number of pairs of delay vectors that are within an $\varepsilon$-neighborhood of each other. For small values of $\varepsilon$ and infinite $N$, $C(\varepsilon) \propto \varepsilon^D$, and the correlation dimension is defined as

$$D = \lim_{\varepsilon \to 0} \lim_{N \to \infty} \frac{\partial \ln C(\varepsilon)}{\partial \ln \varepsilon}. \qquad (4)$$

This dimension will yield the correct integer values for nonfractal objects and is accepted as the best way of estimating a fractal attractor dimension from time-series data. In order to use this algorithm for the practical analysis of a finite time series, we first exclude from consideration all pairs of points that are less than some specified temporal distance from each other; this threshold value is determined from the space-time separation plot. Next, we plot the local slopes of the correlation

sum against the neighborhood $\varepsilon$ on a semilogarithmic scale for several embedding dimensions. If for all embedding dimensions $m > m_0$ there is a region where all the curves saturate at a common plateau, then that value is the correlation dimension. Note that the correlation sum can be computed automatically, while the correlation dimension has to be determined through careful interpretation [20].

We are now in a position to undertake a critical analysis of the "chaotic model" in [48]. As mentioned above, the model produced in the above work is a locally linear approximation, which is unsuitable for predicting the evolution of a chaotic time series more than a very few time steps ahead. In addition, there are significant problems in the methodology of this paper. First, the three datasets used are far too small; the largest is just 136 elements and the smallest a mere 34 elements long. Estimating chaotic invariants from such small datasets is considered scientifically unsound [20]. Second, the correlation dimension is calculated without any consideration of the issues discussed above; no test for stationarity is carried out nor a test for determinism. There is also no attempt to exclude temporal correlations. Thus, the dimension estimates obtained in [48] are highly questionable. In short, no defensible conclusion concerning the presence or absence of chaos in the three datasets examined can be reached.

## V. Experimental Results

In this section, we describe our application of nonlinear analysis to two publicly available sets of software-reliability growth data. We first describe the specific characteristics of each dataset and review previous work concerning them. We then perform a delay reconstruction of the state space for each dataset, and we use the space-time separation plot to investigate the stationarity of each time series. Next, the method of surrogate data is used to check if the datasets appear predominantly deterministic. Our results indicate that the time series both appear to be stationary and deterministic. Finally, we will attempt to estimate a correlation dimension; we find limited evidence of chaotic behavior in both time series.

### A. Software-Reliability Growth Data

Our work involves the analysis of two time series, each of which consists of interfailure times for a single software system. Each element of the time series represents the elapsed time between the current and previous failures. The data are time-ordered, i.e., the interfailure times are recorded in the order in which they actually occurred. These data are "time" series in the sense of an index of event occurrences, rather than the physical time. Elapsed time is the response variable. The noise known to be present in these time series is discretization noise; the data are discretized to integer values, representing some time scale. For both of our time series, the time scale is the nearest day. These two time series are presented in Figs. 1 and 2.

The first dataset (designated ODC1 in the current paper) was collected by IBM during the course of the Orthogonal Defect Classification project, was described in [4], and was made publicly available in [26]. ODC1 consists of 1207 bug reports, each of which includes the date on which the failure was detected. Thus, these data are discretized to the nearest day and are not presented as interfailure times. In order to convert this dataset to interfailure times, we have followed a recommendation in [26] and assumed that failures occur at random times during the day. We used a uniform random number generator to determine the $j$th interfailure time during a single day and then normalized the interfailure times for one day so that they sum to 1.0. We chose to use this technique because it is strikingly similar to a recommendation in [20]: Discretization noise in a times series should be removed by first adding a uniform white noise in the range $[-0.5, 0.5]$ to the signal and then processing the resulting signal with a nonlinear noise-reduction algorithm. Our second dataset also comes from the IBM Orthogonal Defect Classification project, is also described in [4], and archived in [26]. This dataset consists of 2008 bug reports with an attached date of failure, as in ODC1. Once again, we assumed a random time of arrival for bugs during each day. We have designated this dataset as ODC4.

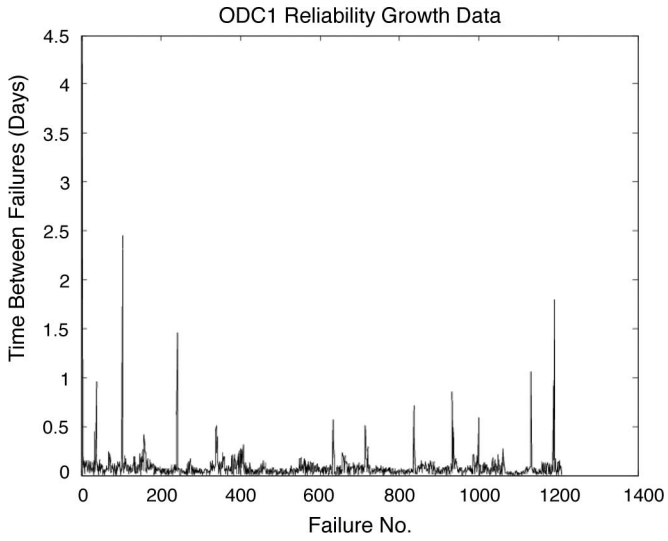



Fig. 1. ODC1 dataset.

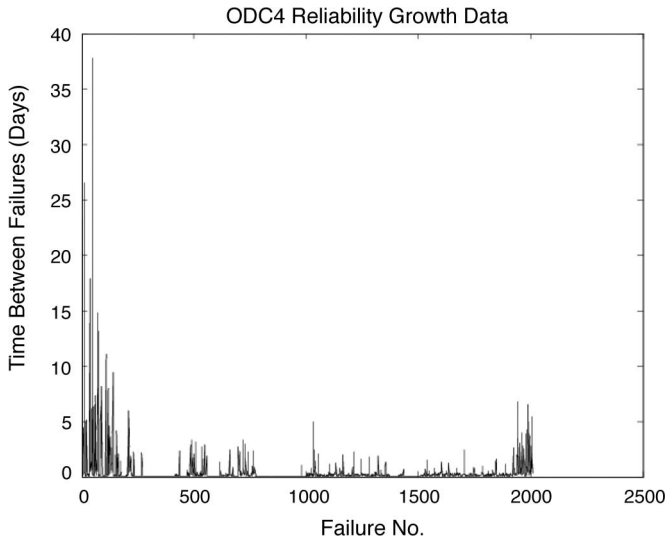



Fig. 2. ODC4 dataset.

TABLE I
MUTUAL INFORMATION AND AUTOCORRELATION VALUES

| v | ODC1-Mut. | ODC1-Aut. | ODC4-Mut. | ODC4-Aut. |
|---|---|---|---|---|
| 1 | 0.0355 | 0.3278 | 0.0212 | 0.2287 |
| 2 | 0.0110 | 0.1008 | 0.0123 | 0.1667 |
| 3 | 0.0716 | 0.0703 | 0.0195 | 0.1744 |
| 4 | 0.0414 | 0.0429 | 0.0146 | 0.1239 |

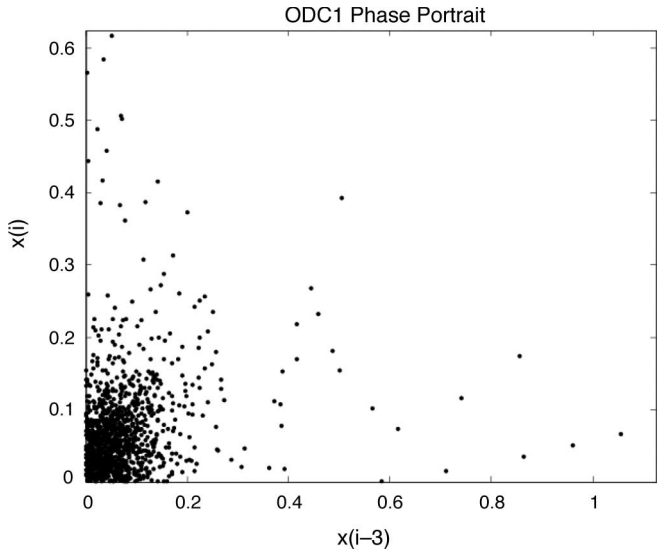



Fig. 3. ODC1 phase portrait.

### B. Nonlinear Analysis

As described in Section IV, the first step in nonlinear analysis is to reconstruct the state space of a system. We have used the method of delay embeddings in this paper, which requires us to select values for embedding dimension $m$ and the embedding delay $\nu$. We begin by searching for appropriate values of $\nu$ using the mutual information statistic and the autocorrelation function in each dataset. Table I contains the results; the first minimum of the mutual information statistic (often a good choice for the delay) occurs at $\nu \leq 4$ for both datasets, but there were no zero crossings within that range for the autocorrelations (often another good choice). We then examined the 2-D phase portraits for each system with delays ranging from one to six. We found that delays of $\nu = 3$ and $\nu = 5$ resulted in the largest apparent structures for ODC1 and ODC4, respectively. Those "best" phase portraits are shown in Figs. 3 and 4, respectively.

When we examine the phase portraits for each dataset, we are able to make the following qualitative statements. In the ODC1 phase portrait, we see some structure on the order of 0.3 units in size, consisting of some fairly sparse trajectories. By contrast, in ODC4, we see a dramatic structure (a double helix) along the $y$ axis that is four units in length. Taking both phase portraits together, we believe that our datasets exhibit qualitative indications of deterministic behavior. We will therefore proceed to reconstruct the state space of each system and determine if the datasets are stationary.

Using the technique of false nearest neighbors, we obtain the results in Fig. 5. For each dataset, we are looking for a value of $m$ such that the ratio of the apparent false nearest

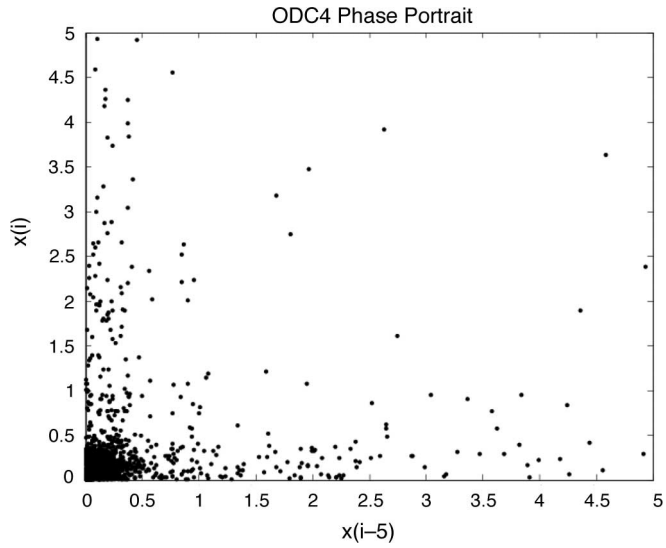



Fig. 4. ODC4 phase portrait.

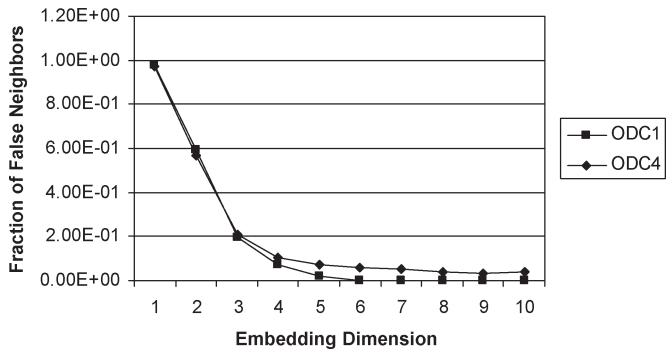



Fig. 5. False nearest neighbor ratios.

neighbors drops to zero. ODC1 reached this point at $m = 6$. ODC4 is a more difficult case since the ratio of false nearest neighbors becomes very small, but never actually reaches zero. A local minimum was observed at $m = 9$. These values are actually quite high; the techniques of nonlinear time-series analysis are most appropriate for low-dimensional chaos (i.e., a fractal dimension of less than 5.0). There is another source of embedding dimension estimates available to us: we can look at the embeddings used in the best of the RBFNs reported in Section VI. After a first pass through the datasets using nonlinear time-series analysis, we then turned our attention to neural-network modeling. The best results we found during that modeling step used embeddings of $m = 3$ and $\nu = 3$ for ODC1, and $m = 2$ and $\nu = 5$ for ODC4. We then reran our nonlinear time-series experiments using these embeddings. In this second run, we found the same evidence of determinism and an improved evidence of chaotic dynamics. These improved results are reported in the remainder of this paper. (Note that, as we stated, nonlinear analysis is less mature than its linear counterpart. Multiple iterations of analysis, using results from the previous analysis to focus and improve later analyses, are typical practice in this domain [20].)

Space-time separation plots for ODC1 and ODC4 are shown in Figs. 6 and 7, respectively. Curves of constant probability for two points separated by a temporal distance of $T$ to be separated

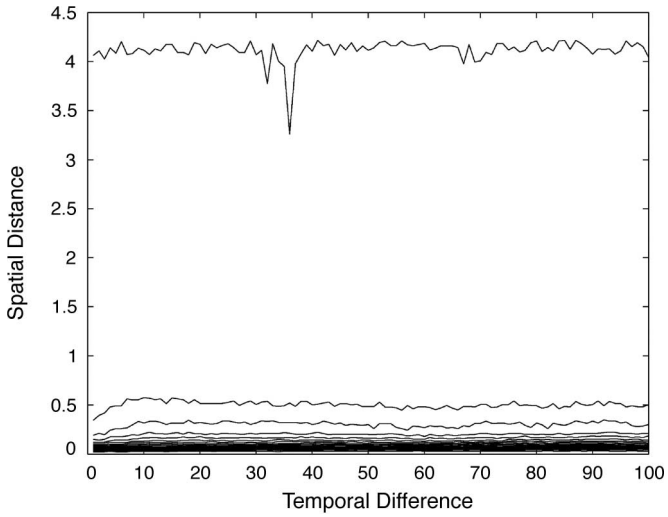



Fig. 6. Space-time separation plot for ODC1.

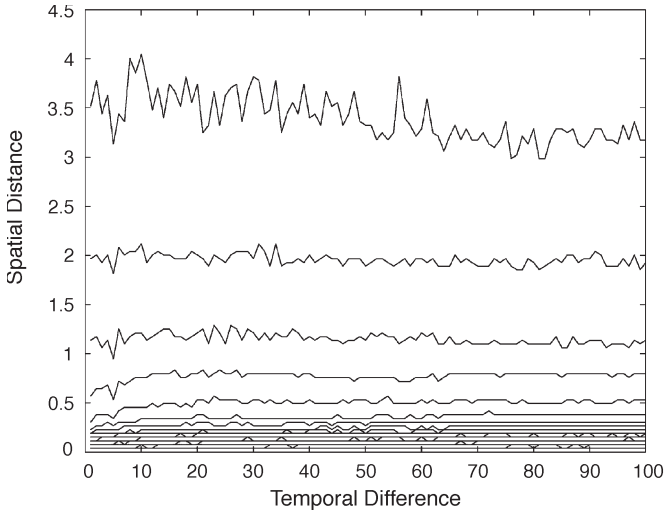



Fig. 7. Space-time separation plot for ODC4.

by a spatial distance of no more than $\varepsilon$ are plotted for probabilities of 0.05, 0.1, 0.15, and so forth. Both datasets appear to be stationary since the curves in each plot all saturate in rough plateaus. We can avoid temporal correlations in estimating the attractor dimension if we exclude points with a temporal distance of less than 35 steps in ODC1 and 70 time steps in ODC4. It may seem odd to discuss stationarity in the context of software debugging; after all, the software is undergoing frequent changes in the reliability growth process. However, we remind the reader that the observable quantity recorded in our time series is the time of discovery for a failure. Considering that these datasets do not include regression testing, it seems quite possible that our time series could be stationary, as the test cases that caused failures previously are not being revisited. Any nonstationarity introduced by debugging is thus not observed.

We have used the method of surrogate data described in [20] to test both of our datasets for the presence of deterministic behavior. We began by attempting to fit 22 analytic probability distributions to each dataset, including the normal, log-normal,

TABLE II
TEST STATISTIC VALUES

| | ODC1 Time Reversal | ODC4 Time Reversal | ODC4 Prediction |
|---|---|---|---|
| Original | -1.31802356 | -2.85292745 | 1.5480 |
| Surr. Min | -0.360812485 | -6.55966377 | 1.6094 |
| Surr. Max | 0.25729689 | 4.96917629 | 1.8481 |

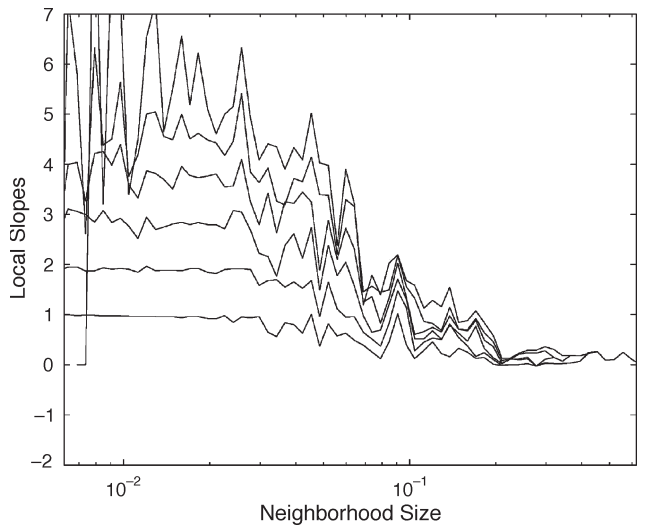



Fig. 8. ODC1 scaling plot.

two-parameter Weibull, two-parameter gamma, exponential, Rayleigh, and beta distributions. Goodness-of-fit was then evaluated using the Kolomogorov–Smirnoff test; at a significance of 0.05, every one of these distributions was rejected for both of our datasets. Thus, no probability structure for the datasets has been found. Hence, in our surrogate data experiments, we used the most general hypothesis in the nonlinear time-series literature: that the data arise from a linear Gaussian process and are distorted through a monotonic nonlinear observation function. We used a two-sided test with a significance of 0.05, which required us to generate 39 surrogates for each dataset [20]. The null hypothesis will be rejected in favor of the alternative (that the data arise from a deterministic process) if the test statistic for the original dataset is maximal or minimal with respect to the surrogates' test statistics. The test statistics used were time-reversal asymmetry and the prediction error from the locally constant projective scheme, as described in [20] and implemented in [14] and [39]. Of these two, the prediction error is considered a more powerful test. The test statistic values for both datasets, as well as the minimum and maximum values for their surrogates, are presented in Table II. By examining this table, we find that we reject the null hypothesis for both datasets. Hence, Table II and our previous attempt to fit a classical distribution to the datasets constitute quantitative evidence that these two datasets are predominantly deterministic in nature.

We have attempted to extract a fractal dimension from the ODC1 and ODC4 datasets using the correlation dimension algorithm. Fig. 8 is a scaling plot for ODC1. Each curve relates the local slopes of the correlation integral to the neighborhood size for a specific embedding dimension. We are looking for a range of values in which all curves representing an embedding

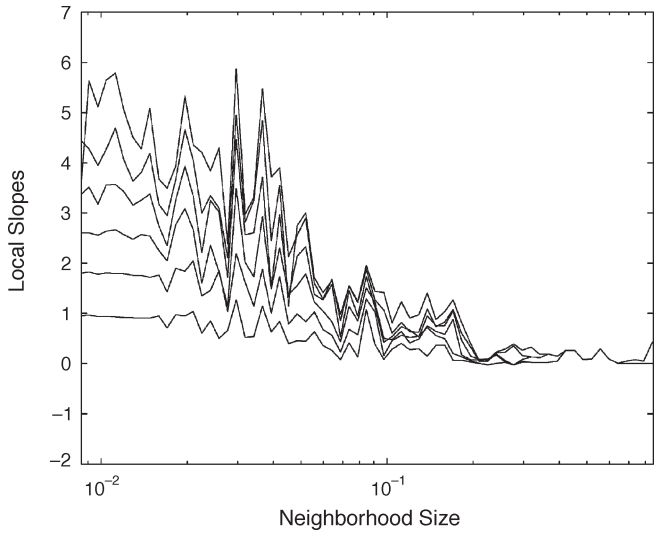



Fig. 9. ODC1 scaling plot after noise reduction.

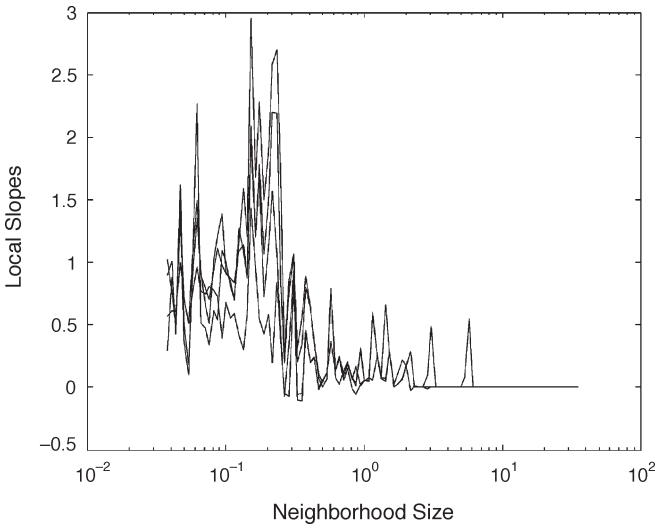



Fig. 10. ODC4 scaling plot.

dimension of $m_0$ or greater saturate at a common plateau; the $y$ value of this plateau would be an estimate of the fractal dimension. The value of $m_0$ is specific to each dataset. By examining Fig. 8, we notice that the curves all have an individual plateau at around $10^{-2}$ units in scale. However, these plateaus represent noise effects, which expand to fill every dimension of a phase space. (That is why each curve seems to represent an increment in the estimated fractal dimension.) What we are looking for is a saturation of the curves at a common plateau. There is a bare hint of this behavior around the $10^{-1}$ unit scale. The use of a nonlinear noise-reduction scheme greatly amplifies this behavior, as in Fig. 9. It is now possible to see a small range just below $10^{-1}$ units where the curves saturate in an oscillation; while not a perfect plateau, this behavior does hint at an attractor dimension between 1 and 1.5, roughly speaking. In ODC4, it is quite difficult to see any plateau behavior in the original time series (see Fig. 10). However, if we look at the scaling plot after nonlinear noise reduction (see Fig. 11), there is a hint of a saturated level oscillation right around $10^{-1}$ units in size, hinting at a dimension of just under one (the dotted

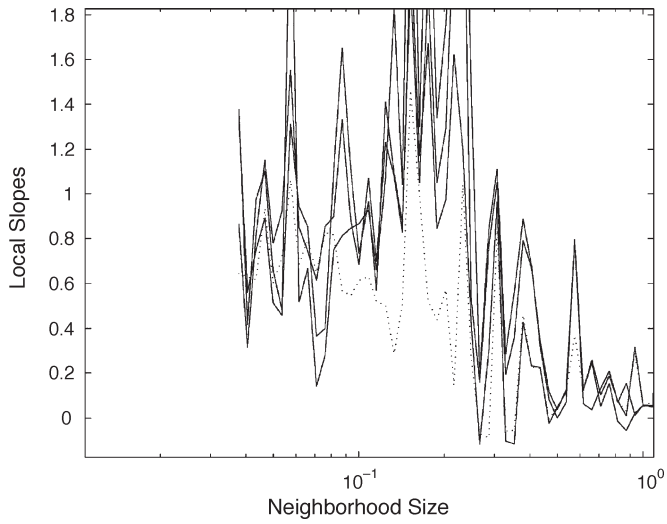



Fig. 11. ODC4 scaling plot after noise reduction.

line is $m = 1$; the others represent $m = 2, 3,$ and $4$). What we can conclude from these four figures is that there is some evidence of chaotic behavior in the two time series, although the evidence is not clear enough for a definitive conclusion that chaotic dynamics are present.

## VI. Predictive Modeling

As stated in [20], the true test of any new model is whether or not that model can produce better forecasts than the previous model. In this section, we describe a series of predictive modeling experiments we have carried out over our two datasets. We have compared RBFNs against several well-known SRMs. These include the Jelinski–Moranda [18], Musa Basic [29], Musa–Okumoto [31], Littlewood–Verrall [23], [25], geometric [28], Schneidewind [38], Yamada S-shaped [45], and the Brooks–Motley [2] models. The RBFN is a trainable neural network, which when trained, acts as a deterministic function approximator. We have used RBFNs as our deterministic model rather than attempting to empirically deduce an explicit chaotic model for these datasets; as discussed in Section III, empirically deducing a chaotic model is not possible, whereas RBFNs can, in fact, yield good predictions of even chaotic time series for a limited time horizon [20].

There are two basic forms of software-reliability-modeling experiments. The first is the descriptive or goodness-of-fit experiment, in which models are compared over the points used to fit the models; this is also known as the in-sample prediction error. This type of experiment reveals how well a model matches the data used to estimate the model parameters, but gives no insight into how the model will perform on new inputs. In the terminology of machine learning, the in-sample error does not reveal how the model will generalize to new situations. The second type of experiment, the predictive experiment, measures the aggregate difference between a model's predictions and a set of inputs that was held back during the estimation of the model; in other words, this is the out-of-sample prediction error. This experimental design requires the available data to be segregated into training and testing sets;

the model will be estimated using the training data only, and its predictive accuracy will be determined using the testing data only. In the specific case of time-series prediction experiments (usually known as forecasting), time dependences in the data are preserved by selecting some number $K$ of the final elements of the time series to be the test set, while the training set consists of the rest of the time series. The experimental results in this section are all predictive modeling experiments [33], [47].

Our experimental design is the generally accepted leave-$k$-elements-out prediction experiment, in which all but the last $k$ elements of a time series are used to train or estimate a model, and then, those final $k$ points are used to measure the prediction error of the model. (This is a more general version of the leave-one-out experiment.) In our experiments, we have chosen $k = 5$, and our error measure is the rms difference between the five predicted values and the five actual hold-out values. There are two sets of experiments reported: one comparing RBFNs against time-between-failure SRMs such as Musa's model and the other comparing RBFNs against failure-count models such as the Yamada S-shaped model. In the time-between-failure experiments (reported in Section VI-B), the five hold-out values are the next five interfailure times. Likewise, in the failure-count experiments, the five hold-out values are the estimated number of failures in each of the next five periods; these results are presented in Section VI-C. We used the SMERFS$\hat{}$3 package [43] to estimate all of the stochastic models (hereafter referred to as the classical SRMs) and obtain prediction values, while we used the MATLAB Neural Networks Toolbox to train and simulate our RBFNs. We stress that the RBFN results we present in this section are not necessarily the optimal predictive accuracy; however, we found that we were able to improve upon the classic SRMs by 25% or more in each dataset. This improvement is large enough that further optimization of the RBFN models will not provide additional insight into the datasets; the RBFN has clearly outperformed the classic models. In the next section, we provide a brief introduction to RBFN modeling.

### A. RBFNs Review

An RBFN model is considerably different than the classical SRMs. RBFNs are a type of neural network which can be trained to approximate any continuous function over a compact set to an arbitrary degree of accuracy (and are hence known as universal approximators). Training of neural networks proceeds by presenting a sequence of input–output pairings (where the input and output may be scalars or vectors) to the network and updating the connection weights between neurons to minimize the network output error on those patterns. The basic architecture of an RBFN is presented in Fig. 12. The network consists of an input layer, a layer of radial basis function (RBF) neurons, and a layer of linear neurons, with each layer fully connected to the following layer. The role of the input layer is to distribute the components of each input vector to all of the radial basis neurons. The radial basis neurons then implement the transfer function

$$y = e^{-(\|x-c\|\cdot b)} \tag{5}$$

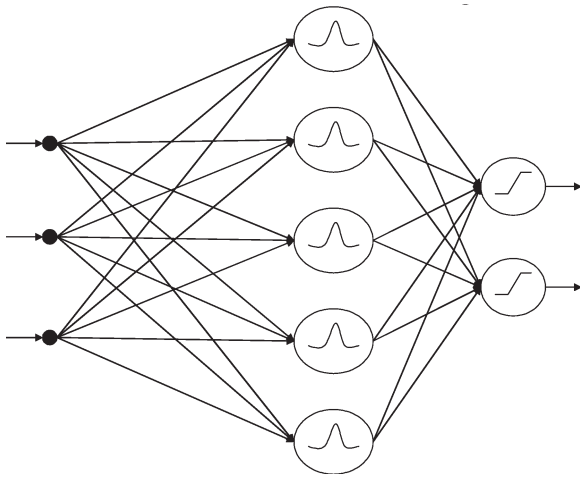


Fig. 12. RBFN.

where $x$ is the network input, $c$ is the center point of the $i$th RBF neuron, and $b$ is a sensitivity parameter that determines the width of the RBF. The RBF neuron measures how similar an input is to its center point $c$, based on the vector distance between them (usually the Euclidean distance). A perfect match results in an output of 1.0 from the neuron, and the output decreases to 0.0 with an increasing distance between $x$ and $c$. The rate of this decrease is controlled by the sensitivity parameter $b$. The linear neurons output a weighted sum of their inputs, and these outputs constitute the components of the network output vector [13].

The training algorithm for RBFN networks, as implemented in the MATLAB Neural Network Toolbox, proceeds by iteratively adding an RBF neuron to the RBF layer and then updating the weight vectors of the linear layer to minimize the sum of squared error for the network over the entire training set. Note that there are always as many linear neurons as network outputs. In the first iteration, there will only be one neuron in the RBF layer. The total sum of squared error obtained by optimizing the weights of the linear neurons is computed and compared with a predefined stopping criteria; assuming that the network error is still greater than the stopping criteria, a new RBF neuron is added. The center of this new neuron is an input pattern in the training set. This process repeats, with each new RBF neuron centered at the input vector that results in the largest decrease in the network error. The process halts when the network error reaches the stopping criterion or when a predefined maximum number of RBF neurons are reached. The ability of the network to correctly respond to unseen input patterns is then measured using an out-of-sample prediction [5].

In order to predict the future evolution of a time series, a neural network must be given the current observation and some number of prior observations and be trained to predict the next observation(s) from these values. We must format these inputs and outputs as a vector of inputs and a scalar (vector) output. The process is, in fact, identical to the delay embedding discussed in Section IV. The input vector will be constructed from the time series using an embedding dimension $m$ and a delay $\nu$; the output dimensionality $n$ shall be the number of steps ahead to predict, and the output delay is one. Hence, the

TABLE III
CLASSIC SRMS VERSUS RBFN, ODC1 (#203 ONWARD)

| Model | Accuracy (RMS) |
|---|---|
| Jelinski-Moranda (MLE) | $3.8155 * 10^{-2}$ |
| Musa Basic (MLE) | $3.8042 * 10^{-2}$ |
| Musa Okumoto (MLE) | $3.7763 * 10^{-2}$ |
| Littlewood-Verrall Linear (MLE) | $3.5493 * 10^{-2}$ |
| Littlewood-Verrall Quadratic (MLE) | $3.5835 * 10^{-2}$ |
| Littlewood-Verrall Linear (LSE) | $3.787 * 10^{-2}$ |
| Littlewood-Verrall Quadratic (LSE) | $4.044 * 10^{-2}$ |
| Jelinski-Moranda (LSE) | $3.8846 * 10^{-2}$ |
| Musa Basic (LSE) | $3.8042 * 10^{-2}$ |
| Musa-Okumoto (LSE) | $3.7763 * 10^{-2}$ |
| **RBFN** | $\mathbf{2.7900 * 10^{-2}}$ |

TABLE IV
CLASSIC SRMS VERSUS RBFN, ODC4 (#1004 ONWARD)

| Model | Accuracy (RMS) |
|---|---|
| Jelinski-Moranda (MLE) | 2.5681 |
| Musa Basic (MLE) | 2.5710 |
| Musa Okumoto (MLE) | 2.7294 |
| Geometric | 2.7449 |
| Littlewood-Verrall Linear (MLE) | 2.8116 |
| Littlewood-Verrall Quadratic (MLE) | 2.7551 |
| Littlewood-Verrall Linear (LSE) | 2.7052 |
| Littlewood-Verrall Quadratic (LSE) | 2.5973 |
| Jelinski-Moranda (LSE) | 1.9601 |
| Musa Basic (LSE) | 2.5710 |
| Musa-Okumoto (LSE) | 2.7294 |
| **RBFN** | **1.4333** |

$j$th input/output pattern for both training and testing will be constructed as

$$\text{Input} = \left(x_{j-(m-1)\nu}, x_{j-(m-2)\nu}, \ldots, x_{j-\nu}, x_j\right)$$
$$\text{Output} = \left(x_{j+1}, x_{j+2}, \ldots, x_{j+n}\right). \quad (6)$$

Using this scheme, there will be $N - (m\nu - 1)$ training vectors and one test vector in our experiments, where $N$ is the number of observations in the time series. Optimizing an RBFN for a particular dataset requires a trial-and-error exploration of a parameter space. In this case, the parameters that we have to explore were the number of RBF neurons, spread of each neuron, embedding dimension, and embedding delay. The final RBFN model for a dataset will be the model exhibiting the best performance on the out-of-sample testing set. See the study in [13] for an in-depth discussion on neural-network modeling.

### B. Times-Between-Failure Models

The SMERFS^3 package, the latest version of the SMERFS reliability-modeling software, is used to estimate the classic SRM models and predict the next five interfailure times. However, SMERFS^3 does have one limitation which impacts our experimental setup. SMERFS^3 can only accommodate up to 1000 data points for reliability model estimation. Hence, our experiments are limited to the last 1005 data points in ODC1 and ODC4, with the models being trained on the first 1000 of these elements and the last five being the test set. To ensure that our results are comparable, we have also limited the RBFN models to the same training and test sets. In the classic SRMs, we have used both the maximum-likelihood and least-squares estimation (LSE) of the model parameters where possible. While there was no difference in the estimation results for the Musa Basic and Musa–Okumoto models, we did observe some differences in the Jelinski–Moranda model, as well as in both the linear and quadratic Littlewood–Verral models. In fact, the Jelinski–Moranda model with LSE yielded the lowest rms prediction error of any classic SRM on the ODC4 dataset.

We report the rms prediction error for the classic SRM models and the RBFN model for ODC1 and ODC4 in Tables III and IV, respectively. Plainly, the RBFN models produced a large improvement in predictive accuracy, improving on the best of the classic SRMs by roughly 26% in ODC1 and 27% in ODC4. Note that, due to numerical problems, the geometric model did not converge for the ODC1 dataset and is thus omitted.

TABLE V
FAILURE-COUNT MODELS FOR ODC1

| Model | Accuracy (RMS) |
|---|---|
| Yamada S-Shaped | 10.3867 |
| B-M Binomial | 9.5992 |
| B-M Poisson | 9.5862 |
| Schneidewind Type 2 | 10.8995 |
| **RBFN** | **7.3653** |

TABLE VI
FAILURE-COUNT MODELS FOR ODC4

| Model | Accuracy (RMS) |
|---|---|
| Yamada S-Shaped | 3.6541 |
| Schneidewind Type 2 | 0.8371 |
| Schneidewind Type 3 | 1.2312 |
| **RBFN** | **0.4970** |

### C. Failure Count Models

Conversion of time-between-failure data to failure-count data involves selecting a time period over which to count failure events. As indicated previously, both ODC1 and ODC4 were accurate to the nearest day only, and we assumed a random arrival time for failures within each day. Thus, we simply returned the ODC1 and ODC4 datasets to counts of failures per day (i.e., the intervals are equal at 1.0). This resulted in 103 failure intervals for the ODC1 data and 800 failure intervals for ODC4. Numerical problems arose with several classical SRM models during estimation; thus, we are able to report the results for the Yamada S-shaped model and the Schneidewind treatment types 2 and 3 models for ODC4, and the Yamada S-shaped model, the Schneidewind type 2 model, and the binomial and Poisson Brooks–Motley models for ODC1. In addition, using the full 800 intervals for ODC4 caused numerical problems for the Yamada S-shaped model as well; thus, we are only able to use the last 700 intervals. For ODC4, intervals 101–795 were used to estimate the classic SRMs and train the RBFN, while intervals 796–800 were the hold-out test data. For ODC1, intervals 1–98 were used to estimate the models and train the RBFN, while intervals 99–103 were the hold-out test data. We report the rms errors for each of the classic SRMs, along with the RBFN results, for ODC1 and ODC4 in Tables V and VI, respectively. Again, the RBFN models clearly outperform the classical SRMs in forecasting accuracy by 23% and 40% in ODC1 and ODC4, respectively.

## VII. Discussion

The experimental results in this paper have supported our contention that nonlinear determinism is a viable alternative to stochastic modeling in software reliability. We have determined that both of our datasets show evidence of deterministic behavior (including some evidence of chaotic dynamics) and that both can be predicted more accurately using nonlinear deterministic models than with some well-known stochastic models. However, the datasets themselves are not ideal candidates for nonlinear time-series analysis, and we were not able to conclusively establish the presence of chaotic dynamics. As with any data analysis, these specific conclusions are limited to the datasets that were actually examined. In our view, the proper scientific position of these results is as two first experiments supporting a novel approach; further investigation of this proposal using high-quality data from modern software projects (which tends to be highly confidential information) will obviously be required.

From a practical standpoint, there is also the question of what we gain from this new theory. In truth, there are both positive and negative aspects to viewing software reliability as a chaotic phenomenon. On the positive side, using nonlinear deterministic models should improve the accuracy of short-term reliability growth forecasts. The time to failure will be more accurately predicted for a limited time horizon; [48] asserts (and we agree) that forecasts over a limited time horizon are the most direct way of measuring the current and immediate future reliability of a software system. This should aid in the day-to-day management of the test cycle. On the negative side, long-term empirical predictions for chaotic processes are infeasible; hence, the optimal-cost approach to determining when to stop testing [42] would also be infeasible. Thus, the implications for overall test management would be significant. It might be necessary to use trend analysis instead of the optimal-cost approach. Trend analysis looks at the overall behavior of a time series within some (relatively long) time window [0, $T$], comparing the mean value of the time series $C()$ to time "$t$" with a line between points $C(0)$ and $C(T)$, denoted as $L(t)$. The quantity

$$\int_0^t C(t) - L(t)dt \tag{7}$$

is computed, and a trend change occurs whenever the sign of (7) changes. All other fluctuations in the time series are considered as purely local fluctuations and dismissed as insignificant [19]. This approach can plainly work, even with a chaotic time series; weak chaotic dynamics will be suppressed by this kind of coarse-grained analysis, while very strong chaos (which we have not observed in our experiments) would manifest itself as very frequent trend changes. On observing such a pattern of frequent changes, a project manager would not need to see an accurate predictive model before concluding that the project is in serious trouble. In the case of weaker chaos, the manager would be able to look at long-term trends and base-project decision making (i.e., release or continue testing) on those trends. However, trend analysis does not permit a finely nuanced cost–benefit analysis, such as would be provided by an optimal-cost approach, nor is there an obvious exit criterion from testing when using trend analysis; the manager's judgment plays a much greater role [19].

It is also important to look at how chaotic dynamics might actually arise during software development. In Section III, we have presented a qualitative argument that fault injection is highly irregular in nature, and our experiments have provided quantitative evidence in favor of this argument. What, however, could the actual mechanism be? Our proposal is that we observe deterministic features in software-reliability growth data because the actual distribution of faults in the program follows a fractal pattern. According to the study in [6], [7], and [10], a fault in a software system (i.e., the actual mistake in source code) will be associated with a region of the program's input space that will trigger that fault. This region was dubbed the error crystal of the software system. Observing a failure in a software system thus means that an input applied to the system has intersected with an error crystal and the resulting disruption to the program state has propagated all the way to the system output. Another trend that was uncovered is that some error crystals were larger (i.e., consumed a greater hypervolume of the input space than others); these larger error crystals were obviously much easier to detect than smaller ones. The researchers also found an apparent inverse power-law relationship between the size of an error crystal and the number of error crystals of equal size; there were exponentially more small error crystals than large ones. This latter statement is particularly important; one of the signature characteristics of a fractal set is that there are numerous elements of different sizes that make up the set and that there must be an inverse power-law relationship between the size of an element and the number of elements in the set of that size [8]. We hypothesize that the union of all error crystals within the input space of a software system forms a fractal subset of the input space. Note that this hypothesis builds upon the classic input-domain approach in software-reliability modeling.

Testing our fractal-set hypothesis will be a difficult task and is beyond the scope of our current paper. What will be required is to construct the fault set of a modern, large-scale software system. This fault set is the union of all error crystals in the software system. Next, we must determine the topological dimension and Hausdorff dimension for this set. If the topological dimension is strictly greater than the Hausdorff dimension, the set has a fractal geometry. Some of the principal difficulties to be overcome include defining an input space for a modern GUI-driven software system (which we can assume represents some underlying measure space) and mathematical difficulties in determining a Hausdorff dimension for an arbitrary set.

The practical usage of our fractal-set hypothesis would be during the late stages of system testing. Experience has shown that the bugs remaining at this stage of testing tend to be very subtle and complex, and only occur for rare combinations of inputs and environmental conditions. In other words, the error crystals for the remaining faults will be very small and difficult to find. We suggest that using the spatial distribution of known error crystals would be a powerful guide for finding these few unknown error crystals if a global relationship between

the locations of different error crystals can be established. This is precisely what we are suggesting with our fractal-set hypothesis. By using the known fractal distribution of the known error crystals, we may be able to predict where unseen error crystals lie and thus catch the underlying faults before the system is deployed. We expect that this will only be possible at the very end of systems testing because of the power-law relationship at work in fractal sets. A linear improvement in predictive accuracy for a fractal model will most likely require an exponential increase in the available data. Thus, the end of systems test is probably the only point in the testing cycle where there is enough data to support the construction of a fractal model.

## VIII. Summary and Future Work

In this paper, we have advanced chaos theory as an alternative to stochastic models of software reliability. Using two publicly available sets of software-reliability data, we have found evidence for predominantly deterministic behavior rather than stochastic behavior, using the best practices in chaotic time-series analysis. These results have been validated through a predictive modeling experiment, contrasting RBFNs (a popular and powerful deterministic nonlinear modeling tool) against several well-known stochastic SRMs. In our experiments, the RBFNs outperformed the stochastic models by approximately 25% in both datasets. We closed the paper by proposing a causal mechanism to explain our findings: that software faults form a fractal subset of the input space of a program. Our future work in this area will focus on two projects. First, we will attempt to replicate our experimental results on high-quality reliability data obtained from a modern software project. Second, we will attempt to reconstruct the fault set of a large software system, and determine if this fault set is, in fact, a fractal subset of the system input space.

## References


[1] F. P. Brooks, Jr., *The Mythical Man-Month: Essays on Software Engineering*. Reading, MA: Addison-Wesley, 1995. Anniversary Edition.

[2] W. D. Brooks and R. W. Motley, "Analysis of discrete software reliability models," Rome Air Development Center, Rome, NY, Tech. Rep. RADC-TR-80-84, Apr. 1980.

[3] K.-Y. Cai, C.-Y. Wen, and M.-L. Zhang, "A critical review on software reliability modeling," *Reliab. Eng. Syst. Safety*, vol. 32, no. 3, pp. 357–371, 1991.

[4] R. Chillarege, "Orthogonal defect classification," in *Handbook of Software Reliability Engineering*, M. R. Lyu, Ed. New York: McGraw-Hill, 1996, pp. 359–400.

[5] H. Demuth and M. Beale, *Neural Network Toolbox for Use With MATLAB: User's Guide, Version 4*. Natick, MA: MathWorks, 2002.

[6] J. R. Dunham, "Experiments in software reliability: Life-critical applications," *IEEE Trans. Softw. Eng.*, vol. SE-12, no. 1, pp. 110–123, Jan. 1986.

[7] J. R. Dunham and G. B. Finelli, "Real-time software failure characterization," in *Proc. 5th Annu. Conf. Comput. Assurance, COMPASS*, Gaithersburg, MD, Jun. 25–28, 1990, pp. 39–45.

[8] K. Falconer, *Fractal Geometry: Mathematical Foundations and Applications*. New York: Wiley, 1990.

[9] W. Farr, "Software reliability modeling survey," in *Handbook of Software Reliability Engineering*, M. R. Lyu, Ed. New York: McGraw-Hill, 1996, pp. 71–115.

[10] G. B. Finelli, "NASA software failure characterization experiments," *Reliab. Eng. Syst. Safety*, vol. 32, no. 1/2, pp. 155–169, 1991.

[11] M. A. Friedman and J. M. Voas, *Software Assessment: Reliability, Safety, Testability*. New York: Wiley, 1995.

[12] A. L. Goel, "Software reliability models: Assumptions, limitations, and applicability," *IEEE Trans. Softw. Eng.*, vol. SE-11, no. 12, pp. 1411–1423, Dec. 1985.

[13] S. Haykin, *Neural Networks: A Comprehensive Foundation*, 2nd ed. Upper Saddle River, NJ: Prentice-Hall, 1999.

[14] R. Hegger, H. Kantz, and T. Schreiber, "Practical implementation of nonlinear time series methods: The TISEAN package," *CHAOS*, vol. 9, no. 2, pp. 413–435, 1999.

[15] C.-Y. Huang and S.-Y. Kuo, "Analysis of incorporating logistic testing-effort function into software reliability modeling," *IEEE Trans. Rel.*, vol. 51, no. 3, pp. 261–270, Sep. 2002.

[16] C.-Y. Huang, M. R. Lyu, and S.-Y. Kuo, "A unified scheme of some nonhomogenous Poisson process models for software reliability," *IEEE Trans. Softw. Eng.*, vol. 29, no. 3, pp. 261–269, Mar. 2003.

[17] R. K. Iyer and D. J. Rossetti, "Effect of system workload on operating system reliability: A study on IBM 3081," *IEEE Trans. Softw. Eng.*, vol. SE-11, no. 12, pp. 1438–1448, Dec. 1985.

[18] Z. Jelinski and P. B. Moranda, "Software reliability research," in *Proc. Statistical Comput. Performance Eval.*, Providence, RI, Nov. 22–23, 1971, pp. 465–484.

[19] K. Kanoun and J.-C. Laprie, "Trend analysis," in *Handbook of Software Reliability Engineering*, M. R. Lyu, Ed. New York: McGraw-Hill, 1996, pp. 401–437.

[20] H. Kantz and T. Schreiber, *Nonlinear Time Series Analysis*. New York: Cambridge Univ. Press, 1997.

[21] T. M. Khoshgoftaar and R. M. Szabo, "Investigating ARIMA models of software system quality," *Softw. Qual. J.*, vol. 4, no. 1, pp. 33–48, Mar. 1995.

[22] E. E. Lewis, *Introduction to Reliability Engineering*, 2nd ed. New York: Wiley, 1996.

[23] B. Littlewood, "The Littlewood–Verrall model for software reliability compared with some rivals," *J. Syst. Softw.*, vol. 1, no. 3, pp. 251–258, 1980.

[24] ——, "Theories of software reliability: How good are they and how can they be improved?" *IEEE Trans. Softw. Eng.*, vol. SE-6, no. 5, pp. 489–500, Sep. 1980.

[25] B. Littlewood and J. L. Verrall, "A Bayesian reliability model with a stochastically monotone failure rate," *IEEE Trans. Rel.*, vol. R-23, no. 2, pp. 108–114, Jun. 1974.

[26] M. R. Lyu, *Handbook of Software Reliability Engineering*. New York: McGraw-Hill, 1996.

[27] T. A. Mazzuchi and R. Soyer, "A Bayes-empirical Bayes model for software reliability," *IEEE Trans. Rel.*, vol. 37, no. 2, pp. 248–254, Jun. 1988.

[28] P. B. Moranda, "Event-altered rate models for general reliability analysis," *IEEE Trans. Rel.*, vol. R-28, no. 5, pp. 376–381, Dec. 1979.

[29] J. D. Musa, "Validity of execution-time theory of software reliability," *IEEE Trans. Rel.*, vol. 28, no. 3, pp. 181–191, Aug. 1979.

[30] J. D. Musa, A. Iannino, and K. Okumoto, *Software Reliability: Measurement, Prediction, Application*. New York: McGraw-Hill, 1987.

[31] J. D. Musa and K. Okumoto, "A logarithmic Poisson execution time model for software reliability measurement," in *Proc. 7th Int. Conf. Softw. Eng.*, Orlando, FL, Mar. 26–29, 1984, pp. 230–238.

[32] H. Pham, *Software Reliability*. New York: Springer-Verlag, 2000.

[33] H. Pham, L. Nordmann, and X. Zhang, "A general imperfect-software-debugging model with S-shaped fault-detection rate," *IEEE Trans. Rel.*, vol. 48, no. 2, pp. 169–175, Jun. 1999.

[34] L. Pham and H. Pham, "Software reliability models with time-dependent hazard function based on Bayesian approach," *IEEE Trans. Syst., Man, Cybern. A, Syst., Humans*, vol. 30, no. 1, pp. 25–35, Jan. 2000.

[35] ——, "A Bayesian predictive software reliability model with pseudo-failures," *IEEE Trans. Syst., Man, Cybern. A, Syst., Humans*, vol. 31, no. 3, pp. 233–238, May 2001.

[36] A. Provenzale, L. A. Smith, R. Vio, and G. Murante, "Distinguishing between low-dimensional dynamics and randomness in measured time series," *Physica D*, vol. 35, no. 1–4, pp. 31–49, Sep. 1992.

[37] J. A. Rice, *Mathematical Statistics and Data Analysis*. Belmont, CA: Wadsworth, 1995.

[38] N. F. Schneidewind, "Analysis of error processes in computer software," *SIGPLAN Not.*, vol. 10, no. 6, pp. 337–346, Jun. 1975.

[39] T. Schreiber and A. Schmitz, "Surrogate time series," *Physica D*, vol. 142, no. 3/4, pp. 346–382, Aug. 2000.

[40] N. D. Singpurwalla, "Determining an optimal time interval for testing and debugging software," *IEEE Trans. Softw. Eng.*, vol. 17, no. 4, pp. 313–319, Apr. 1991.

[41] N. D. Singpurwalla and R. Soyer, "Assessing (software) reliability growth using a random coefficient autoregressive process and its ramifications," *IEEE Trans. Softw. Eng.*, vol. SE-11, no. 12, pp. 1456–1464, Dec. 1985.
[42] N. D. Singpurwalla and S. P. Wilson, "Software reliability modeling," *Int. Stat. Rev.*, vol. 62, no. 3, pp. 289–317, Dec. 1994.
[43] W. Stoneburner, *SMERFS Downloads*, Jan. 29, 2003. [Online]. Available: http://www.slingcode.com/smerfs/downloads/SlingCode.com
[44] J. Theiler, S. Eubank, A. Longtin, B. Galdrikian, and J. D. Farmer, "Testing for nonlinearity in time series: The method of surrogate data," *Physica D*, vol. 58, no. 1–4, pp. 77–94, Sep. 1992.
[45] S. Yamada, M. Ohba, and S. Osaki, "S-shaped reliability growth modeling for software error detection," *IEEE Trans. Rel.*, vol. R-32, no. 5, pp. 475–478, Dec. 1983.
[46] X. Zhang, X. Teng, and H. Pham, "Considering fault removal efficiency in software reliability assessment," *IEEE Trans. Syst., Man, Cybern. A, Syst., Humans*, vol. 33, no. 1, pp. 114–120, Jan. 2003.
[47] ——, "Considering fault removal efficiency in software reliability assessment," *IEEE Trans. Syst., Man, Cybern. A, Syst., Humans*, vol. 33, no. 1, pp. 114–120, Jan. 2003.
[48] F.-Z. Zou and C.-X. Li, "A chaotic model for software reliability," *Chin. J. Comput.*, vol. 24, no. 3, pp. 281–291, Mar. 2001.

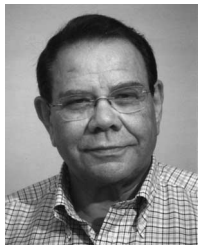
**Scott Dick** (S'97–M'02) received the B.Sc., M.Sc., and Ph.D. degrees from the University of South Florida, Tampa, in 1997, 1999, and 2002, respectively. His Ph.D. dissertation received the USF Outstanding Dissertation Prize in 2003.

He has been an Assistant Professor with the Department of Electrical and Computer Engineering, University of Alberta, Edmonton, AB, Canada, since July 2002. He has published more than a dozen scientific articles in journals and conference proceedings.

Dr. Dick is a member of the Association for Computing Machinery and the American Society for Engineering Education. He is an Associate Member of Sigma Xi.

**Cindy L. Bethel** received the B.S. degree in computer science from the University of South Florida, Tampa, in 2004, where she is currently working toward the Ph.D. degree in computer science and engineering.

She is an NSF Graduate Research Fellow, with her research focus in the areas of artificial intelligence, robotics, and affective computing.



**Abraham Kandel** (S'68–M'74–SM'79–F'92) received the B.Sc. degree in electrical engineering from Technion—Israel Institute of Technology, Haifa, the M.S. degree in electrical engineering from the University of California, Santa Barbara, and the Ph.D. degree in electrical engineering and computer science from the University of New Mexico, Albuquerque.

He is a Distinguished University Research Professor and the Endowed Eminent Scholar in computer science and engineering at the University of South Florida and is the Executive Director of the National Institute for Systems Test and Productivity. He was the Chairman of the Computer Science and Engineering Department at the University of South Florida (1991–2003) and the Founding Chairman of the Computer Science Department at the Florida State University (1978–1991). He has published over 500 research papers for numerous professional publications in computer science and engineering. He is also the author, coauthor, editor, or coeditor of 46 text books and research monographs in the field. He is a member of the editorial boards of leading international journals.

Dr. Kandel is a Fellow of the Association for Computing Machinery, New York Academy of Sciences, American Association for the Advancement of Science, and the International Fuzzy Systems Association. He was awarded the Fulbright Senior Research Fellow Award at Tel-Aviv University in 2003–2004. In 2005, he was selected by the Fulbright Foundation as a Fulbright Senior Specialist in applied fuzzy logic, computational intelligence, data mining, and related fields.