Measuring and Predicting the Linux Kernel Evolution

Francesco Caprio*, Gerardo Casazza**, Massimiliano Di Penta*, Umberto Villano*
frac@unisannio.it, gec@unisannio.it, dipenta@unisannio.it, villano@unisannio.it

(*)University of Sannio, Faculty of Engineering - Piazza Roma, I-82100 Benevento, Italy
(**)University of Naples “Federico II”, DIS - Via Claudio 21, I-80125 Naples, Italy

Abstract

Software systems are continuously subject to evolution to add new functionalities, to improve quality or performance, to support different hardware platforms and, in general, to meet market request and/or customer requirements.

As a part of a larger study on software evolution, this paper proposes a method to estimate the size and the complexity of a software system, that can be used to improve the software development process. The method is based upon the analysis of historical data by means of time series.

The proposed method has been applied to the estimation of the evolution of 68 subsequent stable versions of the Linux kernel in terms of KLOCs, number of functions and average cyclomatic complexity.

Keywords: Source code metrics, time series, prediction, Linux kernel

1. Introduction

It is widely recognized that software systems must evolve to meet user ever-changing needs [9, 10]. Several driving factors for evolution may be identified: these include new functionalities added, lack of software quality, lack of overall system performance, software portability (on new software and hardware configurations, i.e., new platforms) and market opportunities.

As a part of a larger study, we are investigating the influence of software evolution on its size. In this paper a method to estimate size and complexity of the next release of a software system is proposed. The method is based upon the analysis via *time series* of historical data.

The principal reason for estimating both size and complexity of a software product is to help the developing process. The quality of a software development plan strongly depends on the quality of the estimated size and complexity. Humphrey recognized a poor size estimate as one of the principal reasons of projects failure [8]. Both size and complexity are accurate predictors of the resources required to the development of a software product. For example, the relation between size, complexity and required resources has led to the development of a number of cost models such as COCOMO and Slim [1, 14]. These models require as input an estimate of size and complexity, and the resulting accuracy of the model is related to the accuracy of the estimated input. Several studies show size estimate errors as high as 100%, which imply poor resources estimation and unrealistic project scheduling [7].

A time series is a collection of observations made sequentially in time. One of the possible objectives in analyzing time series is *prediction*: given an observed time series, it is possible to predict its future values. The prediction of future values requires the identification of a model describing the time series dynamics.

Given the number of releases of a software system, the related sequence of sizes can be thought of as a time series. Once the time series has been modeled, it is possible to predict the size of the future releases.

To investigate such a conjecture, the source code of 68 subsequent stable releases of the Linux kernel, ranging from version 1.0 up to version 2.4.0, was downloaded from http://www.kernel.org. Its size and complexity in terms of LOC (lines of code), number of functions and average cyclomatic complexity was evaluated, in order to obtain the time series describing the size evolution. Then, the dynamics behind the Linux kernel evolution was modeled using time series. Finally, a cross-validation procedure was performed to assess the accuracy of the estimates thus produced.

2. Background Notions

A time series is a collection of observations made sequentially in time; examples occur in a variety of fields, ranging from economics to engineering.

Time series can be modeled using *stochastic processes* [12]. A stochastic process can be described as a statistical phenomenon that evolves in time according to probabilistic laws. Mathematically, it may be defined as a collection of random variables ordered in time and defined at a set of time points which may be continuous or discrete.

One of the possible objectives in analyzing time series is *prediction*: given an observed time series, one may want to predict its future values. The prediction of future values requires the identification of a model describing the time series dynamics. There are many classes of time series models to choose from; the most general is the ARIMA class, which includes as special cases the AR, MA and ARMA classes.

A discrete-time process is a purely random process if it consists of a sequence of random variables $\{Z_t\}$ which are mutually independent and identically distributed. By definition, it follows that purely random processes have constant mean and variance.

Under the assumption that $\{Z_t\}$ is a discrete purely random process with mean zero and variance σ_Z^2, a process $\{X_t\}$ is said to be a moving average process of order q (MA(q)) if

$$X_t = Z_t + \beta_1 Z_{t-1} + \ldots + \beta_q Z_{t-q} \tag{1}$$

where $\{\beta_i\}$ are constants. Once the backward shift operator B has been defined as

$$B^j X_t = X_{t-1} \tag{2}$$

a moving average process can be written as

$$X_t = \Theta(B) Z_t \tag{3}$$

where

$$\Theta(B) = 1 + \beta_1 B + \ldots + \beta_q B^q \tag{4}$$

If $\{Z_t\}$ is a discrete purely random process with mean zero and variance σ_Z^2, then a process $\{X_t\}$ is said to be an autoregressive process of order p (AR(p)) if

$$X_t = \alpha_1 X_{t-1} + \ldots + \alpha_p X_{t-p} + Z_t \tag{5}$$

where $\{\alpha_i\}$ are constants. This is similar to a multiple regression model, where $\{X_t\}$ is not regressed on independent variables but on past variables of $\{X_t\}$.

Broadly speaking, a MA(q) explains the present as the mixture of q random impulses, whereas an AR(p) process builds the present in terms of the past p events. A useful class of models for time series is obtained by combining MA and AR processes.

A mixed autoregressive moving-average process containing p AR terms and q MA terms is said to be an ARMA process of order (p, q). It is given by

$$X_t = \alpha_1 X_{t-1} + \ldots + \alpha_p X_{t-p} + Z_t + \beta_1 Z_{t-1} + \ldots + \beta_q Z_{t-q} \tag{6}$$

where $X(t)$ is the original series and $Z(t)$ is a series of unknown random errors which are assumed to follow the normal probability distribution.

Using the backward shift operator B, the previous equation may be written in the form

$$\Phi(B) X_t = \Theta(B) Z_t \tag{7}$$

where

$$\begin{aligned} \Phi(B) &= 1 - \alpha_1 B - \ldots - \alpha_p B^p \\ \Theta(B) &= 1 + \beta_1 B + \ldots + \beta_q B^q \end{aligned} \tag{8}$$

A time series is said to be strictly stationary if the joint distribution of $X(t_1) \ldots X(t_n)$ is the same as the joint distribution of $X(t_1 + \tau) \ldots X(t_n + \tau)$ for all $t_1, \ldots t_n, \tau$. In other words, shifting time origin by an amount τ has no effects on the joint distributions, which must therefore depend on the intervals between $t_1, t_2, \ldots, t_n$.

The importance of ARMA processes lies in the fact that a stationary time series may often be described by an ARMA model involving fewer parameters than a pure MA or AR process [4]. However, even if stationary time series can be efficiently fitted by an ARMA process [13], most time series are non-stationary.

Box and Jenkins introduced a generalization of ARMA processes to deal with the modeling of non-stationary time series [2]. In particular, if in equation (7) X_t is replaced with $\nabla^d X_t$, it is possible to describe certain types of non-stationarity time series. Such a model is called ARIMA (Auto Regressive Integrated Moving Average) because the stationary model, fitted to the differenced data, has to be summed or *integrated* to provide a model for the non-stationary data. Writing

$$W_t = \nabla^d X_t = (1 - B)^d X_t \tag{9}$$

the general process $ARIMA(p, d, q)$ is of the form

$$W_t = \alpha_1 W_{t-1} + \ldots + \alpha_p W_{t-p} + Z_t + \ldots + \beta_q Z_{t-q} \tag{10}$$

More details on time series can be found in [2].

3. The Model

A wide variety of prediction procedures are proposed by [3, 6, 18]; the method proposed here relies on the Box and Jenkins one [2]: an ARIMA(p, d, q) process includes as special case an ARMA process (i.e., ARIMA(p,0,q) = ARMA(p, q)).

When modeling a time series, attention should be paid to assess whether the time series is stationary or not. If a time series is stationary, it can be modeled through an ARMA(p, q) process, otherwise an ARIMA(p, d, q) is required. A *non-stationary* time series can be described as a time series whose characteristic parameters change over

Release Series	Initial Release	Numb. of Releases	Time to Start of Next Release Series	Duration of Series
0.01	9/17/91	2	2 months	2 months
0.1	12/3/91	85	27 months	27 months
1.0	3/13/94	9	1 month	12 months
1.1	4/6/94	96	11 months	11 months
1.2	3/7/95	13	6 months	14 months
1.3	6/12/95	115	12 months	12 months
2.0	6/9/96	34	24 months	32 months
2.1	9/30/96	141	29 months	29 months
2.2	1/26/99	19	9 months	still current
2.3	5/11/99	60	12 months	12 months
2.4	1/4/01	4	–	still current

Table 1. Linux Kernels Most Important Events

time. Different measures of stationarity can be employed to decide whether a process (i.e., a time series) is stationary or not. In practice, assessing that a given time series is stationary is a very difficult task, unless a closed-form expression of the underlying time series is known. Non-stationarity detection can be reduced to the identification of two distinct data segments that have significantly different statistic distributions. Several tests can be used to decide whether two distributions are statistically different: Student's t-test, F-test, chi-square test and Kolmogorov-Smirnov test [12].

The Box and Jenkins procedure [2] requires three main steps, briefly sketched below:

1. *Model identification.* The observed time series has to be analyzed to see which ARIMA(p, d, q) process appears to be most appropriate: this requires the identification of the p, d, q parameters.

2. *Estimation.* The actual time series has to be modeled using the previously defined ARIMA(p, d, q) process. This requires the estimation of the $\{\alpha_i\}$ and $\{\beta_j\}$ coefficients defined by (8).

3. *Diagnostic Checking.* The residuals (i.e., the differences between the predicted and the actual values) have to be analyzed to see if the identified model is adequate.

With the *Model Identification Step* the d value has to be set taking into account whether the time series is stationary (i.e., $d = 0$) or not (i.e., $d > 0$). On the other hand, the identification of (p, q) parameters can be obtained following the *Akaike Information Criterion (AIC)*: the model with smallest AIC has to be chosen [17]. In the *Estimation* step, after the estimation of the $\{\alpha_i\}$ and $\{\beta_j\}$ coefficients has been carried out, it is possible to predict time series future values.

Finally, in the *Diagnostic Checking* step, the model adequacy can be tested by plotting the residuals: the residuals of a *good* model have to be small and randomly distributed [4].

4. Case Study

Linux is a Unix-like operating system that was initially written as a hobby by a Finnish student, Linus Torvalds [16]. The first Linux version, 0.01, was released in 1991. Since then, the system has been developed by the cooperative effort of many people, collaborating over the Internet under the control of Torvalds. In 1994, version 1.0 of the Linux Kernel was released. The current version is 2.4, released in January 2001, the latest stable release is 2.4.3 (March 31, 2001).

Unlike other Unixes (e.g., FreeBSD), Linux it is not directly related to the Unix family tree, in that its kernel was written from scratch, not by porting existing Unix source code. The very first version of Linux was targeted at the Intel 386 architecture. At the time the Linux project was started, the common belief of the research community was that high operating system portability could be achieved only by adopting a microkernel approach. The fact that now Linux, which relies on a traditional monolithic kernel, runs on a wide range of hardware platforms, from PalmPilots to Sparc, MIPS and Alpha workstations, clearly points out that portability can also be obtained by the use of a clever code structure.

Linux is based on the Open Source concept: it is developed under the GNU General Public License and its source code is freely available to everyone. "Open Source" refers to the users' freedom to run, copy, distribute, study, change and improve the software. An open source software system includes the source code, and it explicitly promotes the source code as well as compiled forms distribution/redistribution. Very often open source code is distributed under artistic license and/or the Free Software Foundation GPL (GNU General Public License). Open source distribution license shall not restrict any party from selling or giving away the software as a component of an aggregate software distribution containing programs from several different sources. Moreover, the license does not prevent modifications and derived works, and must allow them to be distributed under the same terms as the license of the original software.

However, Linux most peculiar characteristic is that is not an organizational project, in that it has been developed through the years thanks to the efforts of volunteers from all over the world, who contributed code, documentation and technical support. Linux has been produced through a software development effort consisting of more than 3000 developers distributed over 90 countries on five continents [11]. As such, it is a bright example of successful distributed engineering and development project.

A key point in Linux structure is modularity. Without modularity, it would be impossible to use the open-source development model, and to let lot of developers work in parallel. High modularity means that people can work cooperatively on the code without clashes. Possible code changes have an impact confined to the module into which they are contained, without affecting other modules. After the first successful portings of the initial 80386 implementation, the Linux kernel architecture was redesigned in order to have one common code base that could simultaneously support a separate specific tree for any number of different machine architectures.

The use of loadable kernel modules, introduced with the 2.0 kernel version [5], further enhanced modularity, by providing an explicit structure for writing modules containing hardware-specific code (e.g., device drivers). Besides making the core kernel highly portable, the introduction of modules allowed a large group of people to work simultaneously on the kernel without central control. The kernel modules are a good way to let programmers work independently on parts of the system that should be independent.

An important management decision was establishing, in 1994, a parallel release structure for the Linux kernel. Even-numbered releases were the development versions on which people could experiment with new features. Once an odd-numbered release series incorporated sufficient new features and became sufficiently stable through bug fixes and patches, it would be renamed and released as the next higher even-numbered release series and the process would begin again.

Linux kernel version 1.0, released in March 1994, had about 175,000 lines of code. Linux version 2.0, released in June 1996, had about 780,000 lines of code. Version 2.4, released in January 2001, has more than 2.5 millions lines of code. Table 1, which is an updated version of the one published in [11], shows the most important events in the Linux kernel development time table, along with the number of releases produced for each development series.

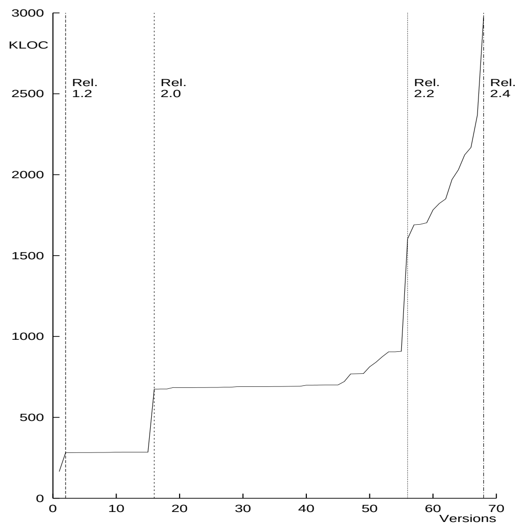

Figure 1. KLOC Evolution

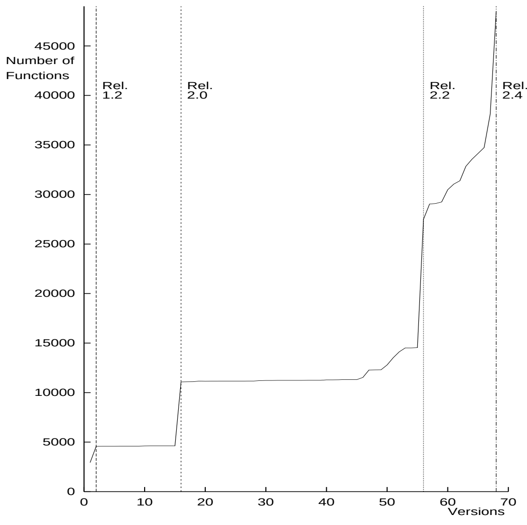

Figure 2. Number of Functions Evolution

5. Predicting Linux Kernel evolution

In order to measure the performance of the presented method the size (in terms of KLOCs, number of functions and average cyclomatic complexity) of 68 subsequent stable releases of the Linux kernel was evaluated. Figg. 1, 2 and 3 show the evolution of Linux kernels ranging from version 1.0 up to 2.4.0. The objective of such experiments

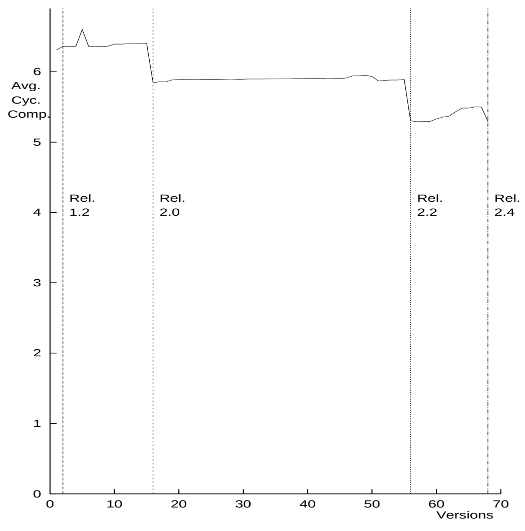

Figure 3. Average Cyclomatic Complexity Evolution

was to test the effectiveness and accuracy of the method on real code.

The experimental activity followed a cross-validation procedure [15]. In each experiment a training time series was extracted from the observed time series. Then, the training time series was analyzed and modeled to predict its future values. Finally, the predicted values were matched against the actual values and the method performance was measured in terms of *absolute prediction error* and *mean absolute prediction error*.

Given a time series, X_t, if $\hat{x}_T$ and x_T are its predicted and actual values at the time T respectively, the absolute prediction error is defined as follows:

$$absolute\,prediction\,error = \frac{abs(x_T - \hat{x}_T)}{x_T} \qquad (11)$$

If ape_k is the absolute prediction error related to the $k-th$ experiment and n is the number of performed experiments, the mean absolute prediction error can be defined as follows:

$$mean\,absolute\,prediction\,error = \frac{1}{n}\sum_{i=1}^{n} ape_i \qquad (12)$$

Given the observed time series, the $k-th$ experiment $(0 \leq k \leq 47)$ was run on the $k-th$ training time series (tts_k), defined as follows[1]:

$$tts_k = \{x_1, \ldots, x_{20+k}\} \qquad (13)$$

[1]For $k = 0$ tts_k contains 20 points.

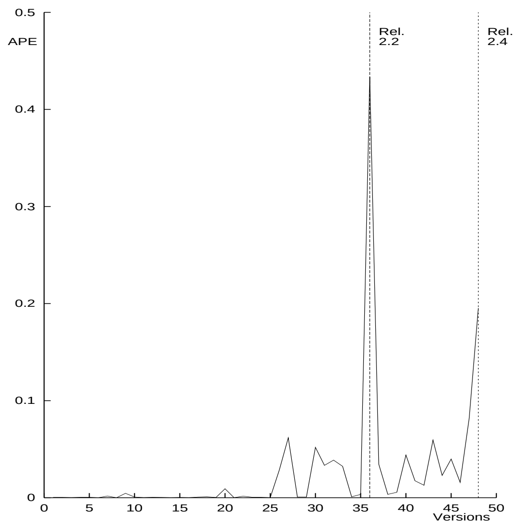

Figure 4. KLOC: APE

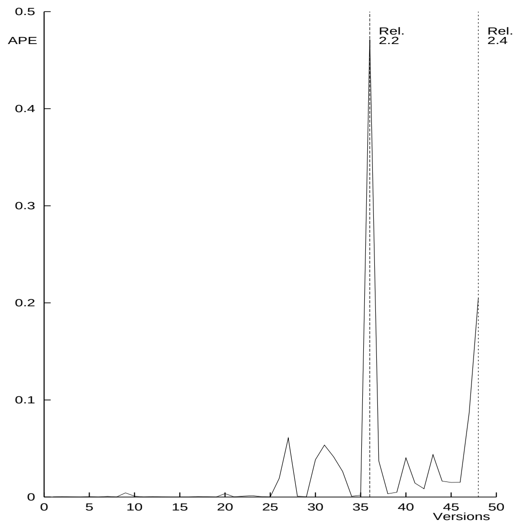

Figure 5. Function Evolution: APE

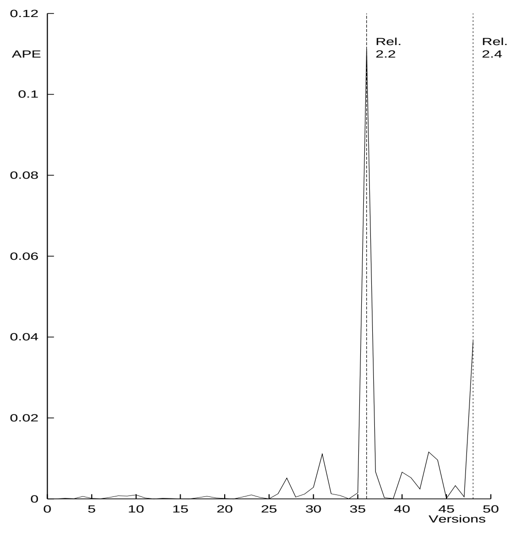

Figure 6. Average Cyclomatic Complexity: APE

where x_1 is the number of KLOC, the number of functions or the average cyclomatic complexity of the Linux kernel 1.0.

During the $k-th$ experiment (i.e., E_k) the one-step-ahead value (i.e., $\hat{x}_{20+k+1}$) was predicted; then, the predicted value was compared against the actual one (i.e., x_{20+k+1}) in order to evaluate both the one-ahead absolute prediction error and the related mean absolute prediction errors.

Figg. 4, 5 and 6 show the trend of the one-step-ahead absolute prediction errors for predicting KLOCs, the number of functions and the average cyclomatic complexity, respectively.

The one-ahead mean absolute prediction errors obtained in the experiments carried out are shown in Table 2.

Step(s) ahead	Mean absolute prediction errors
KLOCs	2.59%
# of Functions	2.54%
Av. Cycl. Complexity	0.47%

Table 2. Mean absolute prediction errors

6. Discussion

As shown in Fig.. 1, 2, and 3, major changes in the Linux kernel occurred with releases 1.2, 2.0, 2.2 and 2.4. However, it is worth noting that such changes were primarily developed and tested along the not-stable Linux kernel releases, and then included in the stable ones.

As a result, the predicted values concerned with releases 2.2.0 and 2.4.0 were affected by a high error. No values could be predicted for the releases 1.2 and 2.0, because they belonged to the initial *training time series* (i.e. tts_k for $k = 0$). In fact, as shown in Figg. 1, 2, and 3, the values concerned with the releases 1.2 and 2.0 are within the first 20 points of each time series.

The APEs related to the release 2.2.0 KLOCs and number of functions were respectively 43% and 47% (see Figg. 4, and 5). However, in correspondence of these errors, there was a 80% increase in terms of KLOCs and a 93% increase in terms of number of functions. On the other hand, the error affecting the average cyclomatic complexity was considerably smaller (about 10%, as shown in Fig. 6).

The reasons behind such non-negligible errors can be explained taking into account that highest errors were obtained when the kernel underwent relevant changes. The most important changes introduced into release 2.2.0 of the Linux kernel, namely:

- several improvements for networking (firewalling, routing, traffic bandwidth management) and TCP stack extensions added;
- new filesystems added, NFS daemon improved;
- sound configuration improved, and support for new soundcards added.

The APEs related to the release 2.2.0 KLOCs and number of functions were both of about 20% and, even if bigger than the mean absolute prediction errors (see table 2), they can be considered as acceptable. The most important changes in release 2.4.0 can be summarized as follows:

- memory management improvement (support for addressing over 1 Gbyte on 32 bit processors, new strategy for paging, improvement of virtual memory management);
- raw I/O support, RAID system improvement, Logical Volume Manager introduced;
- multiprocessor support improvement;
- network layer totally rewritten;
- support for USB configuration improved.

7. Conclusions

The proposed method has been applied to predict the evolution of the Linux kernel. The average prediction errors were generally low and, though with some relevant peaks, they can be considered acceptable, and due to the substantial and non-evolutionary changes occurred in that point.

Future work will be addressed to analyze a sequence of both stable and experimental releases of the Linux kernel, in order to take into account the progressive changes done. Moreover, kernel patches will be also considered, in order to measure the amount of changes with higher accuracy than the simple comparison of size differences between two subsequent releases. Finally, multi-step prediction will be performed, and the method will also be applied to other categories of software systems.

References

[1] B. W. Boehm. *Software Engineering Economics*. Prentice-Hall, Englewood Cliffs, NJ, 1981.

[2] G. Box and M. Jenkins. *Time Series Forecasting Analysis and Control*. Holden Day, San Francisco (USA), 1970.

[3] R. G. Brown. *Smoothing, Forecasting and Prediction*. Prentice Hall, 1963.

[4] C. Chatfield. *The Analysis of the Time Series*. Chapman & HallRc, 1996.

[5] J. de Goyeneche and E. de Sousa. Loadable kernel modules. *IEEE Software*, 16(1):65–71, January 1999.

[6] A. Harvey. *Forecasting, Structural Time Series Models and the Kalman Filter*. Cambridge University Press, 1989.

[7] J. Hihn and H. Habib-agahi. Cost estimation of software intensive projects: a survey of current practice. In *Proceedings of the International Conference on Software Engineering*, pages 13–16. IEEE Computer Society Press, 1991.

[8] W. Humprey. *Managing the Softwarre Process*. Addison Wisley, Reading MA, 1989.

[9] M. M. Lehman and L. A. Belady. *Software Evolution - Processes of Software Change*. Academic Press, London, 1985.

[10] M. M. Lehman, D. E. Perry, and J. F. Ramil. On evidence supporting the feast hypothesis and the laws of software evolution. In *Proc. of the Fifth International Symposium on Software Metrics*, Bethesda, Maryland, November 1998.

[11] J. Moon and L. Sproull. Essence of distributed work: The case of the linux kernel. Technical report, First Monday, vol. 5, n. 11 (November 2000), http://firstmonday.org/.

[12] A. Papoulis. *Probability, Random Variables, and Stochastic Processes*. McGraw-Hill, 1984.

[13] D. Piccolo and C. Vitale. *Metodi Statistici per l'Analisi Economica*. Il Mulino, 1989.

[14] L. Putnam and W. Myers. *Measures for Excellence*. Yourdon Press, 1992.

[15] M. Stone. Cross-validatory choice and assesment of statistical predictions (with discussion). *Journal of the Royal Statistical Society B*, 36:111–147, 1974.

[16] L. Torvalds. The linux edge. *Communications of the ACM*, 42(4):38–39, Dec 1999.

[17] W. Venables and B. Ripley. *Modern Applied Statistic with S-PLUS*. Springer, 1999.

[18] M. West and P. Harrison. *Bayesian Forecasting and Dynamic Models*. Springer-Verlag, 1989.